HIGH-PERFORMANCE CORRELATION AND MAPPING ENGINE FOR RAPID GENERATING BRAIN CONNECTIVITY NETWORKS FROM BIG FMRI DATA

A Dissertation

by

JOHN DAVID LUSHER II

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jim Xiuquan Ji
Committee Members,	Peng Li
	Joseph Orr
	Raffaella Righetti
	Suojin Wang
Head of Department,	Miroslav Begovic

August 2018

Major Subject: Electrical Engineering

ABSTRACT

Brain connectivity networks help physicians better understand the neurological effects of certain diseases and make improved treatment options for patients. Voxel-to-Voxel Correlation Analysis (VVCA) of functional magnetic resonance imaging (fMRI) data has been used to create the individual brain connectivity networks. However, an outstanding issue is the long processing time to generate full brain connectivity maps. With close to a million individual voxels, with each having hundreds of samples, in a typical fMRI dataset, the number of calculations involved in a voxel-by-voxel CCA becomes very high. With the emergence of the dynamic time-varying functional connectivity analysis, the population-based studies, and the studies relying on real-time neurological feedbacks, the need for rapid processing methods becomes even more critical. This research dissertation describes a new method which produces high-resolution brain connectivity maps rapidly. This new method accelerates the correlation processing by using an architecture that includes clustered FPGAs and an efficient memory pipeline, which is termed the High-Performance Correlation and Mapping Engine (HPCME). The method has been tested with various datasets from the Human Connectome Project. The results show that HPCME with four FPGAs can improve the VVCA processing speed by a factor of 40 or more over that of a PC workstation with a multicore CPU.

DEDICATION

To my beautiful wife, Jennifer, and lovely daughter, Amelia.

ACKNOWLEDGMENTS

I would like to state my sincerest gratefulness to my committee chair Dr. Jim Ji, who has provided continual support, guidance, friendship, and concern for my well being. Dr. Ji sought opportunities for me, both for my research and to help me broaden my desire to teach. He tutored me in developing my presentations, research articles, and even how to be more active in teaching by watching his example with his students.

Dr. Ji helped direct me into this area of research and helped me grow as an engineer, a teacher, and communicate more effectively. Without his guidance, this dissertation and research would not have been possible.

I also would like to thank Dr. Orr, whose knowledge and experience in neuroscience has helped guide me into a deeper understanding of the field that my research is aimed at improving. Dr. Orr has provided me many hours of counsel, data, and assistance in reviewing articles and my research results. Dr. Orr enabled me mentally take this research from a theoretical concept to a deeper understanding of practical impact in neuroscience.

Also, I would like to thank my wife, Jennifer, and my daughter, Amelia, for supporting me throughout this endeavor. They have sacrificed many days and nights while I was taking classes and otherwise involved in my research. Without them, none of this would have been possible. They collectively encouraged, supported, and stood by me even when I was frustrated and overwhelmed with everything that needed to be done to complete my research.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

Data was provided, in part, by the Human Connectome Project, WU-Minn Consortium (Principal Investigators: David Van Essen and Kamil Ugurbil; 1U54MH091657) funded by the sixteen NIH Institutes and Centers that support the US National Institute of Health (NIH) Blueprint for Neuroscience Research; and by the McDonnell Center for Systems Neuroscience at Washington University.

All other work conducted for this research and dissertation was completed independently.

Funding Sources

This work was supported, in part, by the US National Science Foundation (NSF) under the award number 1606136. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

NOMENCLATURE

ALU	Arithmetic Logic Unit
ANALYZE	Analyze file format developed by the Biomedical Imaging Resource
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface (AMBA Interface)
BIT	BIT File - FPGA Program (Hardware and Firmware)
BRAM	Block RAM
CCA	Cross Correlation Analysis
CSF	Cerebral Spinal Fluid
DDR3	Dual Data Rate 3
DDR4	Dual Data Rate 4
DFC	Dynamic Functional Connectivity
DICOM	Digital Imaging and Communications in Medicine
DMA	Direct Memory Access
DSC	Dice's Similarity Coefficient
DSP	Digital Signal Processing
DWI	Diffusion Weighted Imaging
fMRI	Functional Magnetic Resonance Imaging
fcMRI	Functional Connectivity Magnetic Resonance Imaging
FPGA	Field Programable Gate Array
FLOPS	Floating Point Operations per Second
FWHM	Full Width at Half Maximum

GB	Gigabyte
GPU	Graphics Producing Unit
GTE	Greater Than and Equal
HCP	Human Connectome Project
HPCME	High-Performance Correlation and Mapping Engine
ICA	Independent Component Analysis
ICC	Intrinsic Correlation Contrast
ILA	Integrated Logic Analyzer
LSB	Least Significant Bit
LTE	Less Than and Equal
LVDS	Low Voltage Differential Signaling
MB	Megabyte
MIG	Memory Interface Generator
MRI	Magnetic Resonance Imaging
MSB	Most Significant Bit
NVMe	Non-Volatile Memory Express
NIH	National Institutes of Health (USA)
NITRC	Neuroimaging Informatics Tools and Resources Clearinghouse
PCA	Principal Component Analysis
PCC	Pearson Correlation Coeficient
RAM	Random Access Memory
ROI	Region of Interest
ROM	Read Only Memory
rs-fMRI	Resting State Functional Magnetic Resonance Imaging
VVCA	Seed based Correlation Analysis

SoC	System on Chip
SoM	System on Module
SD	Secure Digital
SVD	Singular-value Decomposition
SPM	Statistical Parametric Mapping
SSD	Solid State Drive
TFLOPS	Terraflops
VDM	Voxel Deviation Matrix
VVCA	Voxel-to-Voxel Correlation Analysis

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

NOMENCLATURE vi

TABLE OF CONTENTS ix

LIST OF FIGURES xi

LIST OF TABLES xv

1. INTRODUCTION 1

    1.1 Research Aims 3
        1.1.1 Proof-of-Concept HPCME Hardware Platform 4
        1.1.2 HPCME System 4
        1.1.3 Validation of HPCME System - HCP Processing and Analysis 5
    1.2 fMRI, rs-MRI, and the Human Connectome Project 6
    1.3 Voxel-to-Voxel Correlation Analysis (VVCA) and Connectedness 8
    1.4 CONN: Functional Connectivity Toolbox 10
    1.5 GPU-PCC Technique 12

2. METHODOLOGY 13

    2.1 Architecture of HPCME 13
    2.2 Architecture of Node Degree Engine 16
    2.3 Preprocessing of HCP Data 26
    2.4 Memory Organization 34
    2.5 r vector and Results 36
    2.6 Postprocessing Steps 38

3. IMPLEMENTATION OF SINGLE PROOF-OF-CONCEPT NODE DEGREE ENGINE .. 41

    3.1 Architecture and Implementation for PoC System 41
    3.2 Logic and Firmware Implementation 45

3.3 Results of Initial Tests 47
3.4 Initial HCP Test and Discussion 50

4. IMPLEMENTATION OF HPCME SYSTEM 52

4.1 Architecture and Implementation System 52
4.2 Logic and Firmware Implementation 56
4.3 Initial HCP Test and Discussion 74

5. VALIDATION OF HPCME SYSTEM 75

5.1 Introduction 75
5.2 Results of HPCME Processing HCP Datasets 77
5.3 PCC Results via CPU 88
5.4 Results of GPU-PCC Method 93
5.5 Results of CONN Toolbox 98

6. DISCUSSION 103

6.1 HPCME 103
6.2 Comparison to GPU Methods 105
6.3 CPU Comparison 106
6.4 CONN Toolkit 107

7. CONCLUSION 109

REFERENCES 110

APPENDIX A. DETAILED RESULTS 115

APPENDIX B. NODE DEGREE ENGINE - SCHEMATICS AND PCB LAYOUT 140

APPENDIX C. NODE DEGREE ENGINE - FPGA DESIGN 161

APPENDIX D. SCRIPTS, SOFTWARE, AND FIRMWARE 180

LIST OF FIGURES

FIGURE Page

1.1 fMRI HCP dataset from test subject #100307 6

1.2 Example of node degree distribution for 0.63% completeness 8

1.3 CONN Toolbox Application Interface 11

2.1 Fundamental HPCME Architecture
https://www.sciencedirect.com/science/article/pii/S1877750317310633 14

2.2 HPCME Processing Data Flow 16

2.3 NDE SoC FPGA General Architecture 17

2.4 NDE Multiplication - Covariance Computation Block Diagram 18

2.5 Seed BRAM Address Generator - State Machine 19

2.6 Accumulator Reset - State Machine 20

2.7 NDE Accumulator - Covariance Computation Block Diagram 21

2.8 NDE Correlation Coefficient Calculation and r Vector Generation 22

2.9 r-Vector Results Processing (C#) 23

2.10 Layout of Logic for NDE System Architecture 24

2.11 Steps for Preprocessing HCP datasets for use on the HPCME and NDEs 26

2.12 Preprocessing Step 1 through 3 (MATLAB Script) 27

2.13 Preprocessing Step 4: Compute voxel statistics (MATLAB Script) 28

2.14 Preprocessing Step 5: Pad voxel data (MATLAB Script) 28

2.15 Preprocessing Step 6: Build voxel data files (MATLAB Script) 30

2.16 Preprocessing Step 7: Build seed data files (MATLAB Script) 31

2.17 Upper Triangle Matrix - Correlation Matrix 32

2.18 Preprocessing Step 8: Build task scripts (MATLAB Script) 33

2.19 Voxel memory organization 35
2.20 Seed memory organization 36
2.21 Correlation r Vector (Result Vector) 37
2.22 r-Vector Results Processing (C#) 38
2.23 Postprocessing connectedness data (MATLAB Script) 40
3.1 Xilinx KCU105 Evaluation Board 41
3.2 Top design layer of interface PCB in Altium Designer and Assembled Interface PCB 42
3.3 Prototype of the NDE 44
3.4 Xilinx Zynq 7Z030 implementation results of 64 Core NDE System 45
3.5 Xilinx Zynq 7Z030 resource utilization of 64 Core NDE System 46
3.6 Initial PoC NDE - logic block diagram 46
3.7 Logic analyzer - development testing of initial core 47
3.8 Total processing and storage with cumulative seed groups for a data set of $128 \times 91 \times 91 \times 1024$ 48
3.9 Individual processing and storage time for each seed group 49
3.10 Initial NDE PoC connectedness map of HCP Subject #100307 51
4.1 The HPCME System - Block diagram layout 52
4.2 The HPCME System - Implementation 53
4.3 HPCME interface and control software 54
4.4 HPCME NDE postprocessing of connectedness data (MATLAB Script) 55
4.5 Xilinx Zynq 7Z030 Implementation Results of 32 Core NDE System 56
4.6 Xilinx Zynq 7Z030 Resource Utilization of 32 Core NDE System 57
4.7 HPCME NDE - logic block diagram 57
4.8 HPCME NDE - Timing diagram - Results write to BRAM (ILA Logic Analyze) 58
4.9 HPCME NDE - Logic schematic (Xilinx Vivado Block Diagram) 59
4.10 HPCME NDE - VHDL: Ports and Core Configuration Parameters 60

4.11 HPCME NDE - VHDL: Floating Point Multiplier and Accumulator 61

4.12 HPCME NDE - VHDL: Floating Point Divider and Comparators 62

4.13 HPCME NDE - VHDL: Result BRAM data state machine 63

4.14 HPCME NDE - VHDL: Result BRAM address state machine 64

4.15 HPCME NDE - VHDL: Result BRAM write enable state machine 65

4.16 HPCME NDE - VHDL: Correlation Coefficient Calculation state machine 66

4.17 HPCME NDE - VHDL: Correlation Coefficient Calculation floating point assignments 67

4.18 HPCME NDE - VHDL: Correlation Coefficient Calculation result capture 68

4.19 HPCME NDE - VHDL: Covariance Core - Multiply and Accumulators 69

4.20 HPCME NDE - VHDL: Covariance Core - Covariance and Standard Deviation capture registers 70

4.21 HPCME NDE - VHDL: Covariance Core - Accumulator reset and counter control state machine 1 of 2 71

4.22 HPCME NDE - VHDL: Covariance Core - Accumulator reset and counter control state machine 2 of 2 71

4.23 HPCME NDE - VHDL: Covariance Core - AXI data stream synchronization 72

5.1 Results file size for HCP datasets with r=0.63 78

5.2 Results file size for HCP datasets with r=0.70 79

5.3 3D representation of HCP Subject 102816 with r=0.63 81

5.4 Connectedness of HCP Subject 102816 with r=0.63 (4-slices) 82

5.5 Connectedness of HCP Subject 102816 with r=0.63 83

5.6 Connectedness of HCP Subject 112516 with r=0.70 84

5.7 Graph of processing time for HCP subjects with r=0.63 87

5.8 CPU Processing - 2.17% complete at 37 hours 89

5.9 CPU implementation of the correlation algorithm 90

5.10 GPU-PCC performance with synthetic fMRI data (GTX980, M=300) 95

5.11 GPU-PCC performance with synthetic fMRI data (GTX980, M=1200) 97

5.12 CONN to HPCME comparison - side-by-side 99

5.13 CONN to HPCME comparison - overlapping 101

6.1 HPCME data processing - breakdown by task (overhead) 104

LIST OF TABLES

TABLE Page

5.1 Results file size for HCP datasets comparison 80

5.2 Processing time for HCP subjects with r=0.63 85

5.3 Processing time for HCP subjects with r=0.70 86

5.4 CPU processing time from published research. Extrapolation to HCP datasets used in this research 92

5.5 GPU-PCC performance with synthetic fMRI data (GTX980, M=300) 93

5.6 GPU-PCC performance with synthetic fMRI data (GTX980, M=1200) 95

5.7 CONN Processing Time 99

5.8 CONN to HPCME comparison - Dice's coefficient 102

6.1 HPCME vs. CPU Methods 107

6.2 CONN to HPCME comparison - Dice's coefficient 108

1. INTRODUCTION

Neuroimaging methods and analyses have rapidly evolved over the last decade, spurred by large collaborative funding initiatives such as the Human Connectome Project (HCP) in the US and the Human Brain Project in the EU. A primary focus of these efforts has been mapping and interpreting the connectivity of the brain. Understanding the wiring of the brain offers exciting possibilities to understand brain functions and aid in early detection and treatment of neurological diseases such as Alzheimer's, addiction, schizophrenia, dyslexia, autism, and ADHD [1]. A benefit of these initiatives is that much of the data is made freely available to scientists, fostering discovery and supporting broad and varied approaches towards connectomics. One such dataset, the WU-Minn-Oxford HCP contains high-resolution, high-quality Functional Magnetic Resonance Imaging (fMRI), resting state-fMRI (rs-fMRI), diffusion-weighted imaging (DWI) data, and structural imaging, from a relatively healthy, normative community population [2, 3, 4].

Processing and analyzing these large datasets to determine the correlation between regions of the brain and functional tasks to generate a neurological connectivity map is computationally intensive. Creating these maps is particularly challenging when brain parcellation is a data-driven approach rather than a method that uses simplified atlases, or when dynamic functional connectivity (DFC) is used to investigate time-varying functional interactions among a large number of nodes [5]. While the HCP datasets are available in a preprocessed format, there are situations where researchers may wish to analyze raw data or conduct additional or alternative processing. Thus requiring researchers to have access to High-Performance Computing resources, which may be prohibitive due to a lack of access, high shared loads, and high costs. The same challenge also presents when a large number of subjects are involved in a study [6], or when real-time neurological feedbacks are needed [7, 8].

The dominant neuroscience techniques for building connectivity maps are fMRI (Functional Magnetic Resonance Imaging), resting state fMRI (rs-fMRI), and DWI (Diffusion-Weighted Imaging). fMRI-based connectivity creates connectivity maps based upon the statistical correlation be-

tween in-scanner behavioral functions or tasks, and fluctuations in brain activity. These can be motor movement, gambling, social interaction, talking, emotional response, and various others. rs-fMRI, on the other hand, does not require in-scanner tasks, and instead, builds connectivity maps by examining intrinsic connectivity between voxels or nodes of the human brain, thus permitting the mapping of multiple functional networks which correlate with those observed in task-based fMRI [9, 10, 11, 12]. However, in all of the methods mentioned above, there is no agreed-upon standard for defining the resolution (i.e., the voxel-to-voxel level or region-to-region level, with regions varying greatly in their possible size) at which these networks should be defined [13].

The research presented here describes the High-Performance Correlation and Mapping Engine (HPCME) that enables high throughput processing of brain connectivity datasets. The HPCME system is a computer workstation and a high-performance FPGA (Field Programmable Gate Array) co-processing engine optimized for computing high volume correlation data. The HPCME system demonstrates the ability to generate voxel-to-voxel brain network connectivity maps within seven hours or less from the high-resolution HCP datasets. This research was focused on three specific aims: (1) Design and build a proof-of-concept HPCME hardware platform; (2) Expand the capability of the HPCME system to achieve the targeted performance objectives; and (3) Validate and demonstrate the HPCME with various datasets from the HCP and compare the results generated by using existing connectivity toolkits.

This research will show that the HPCME can be a useful tool to rapidly process the brain connectome data at finer resolutions, which potentially could lead to discoveries in diagnosis and treatment of neurological diseases and cognitive degradation. This dissertation will compare the results from an implementation of the HPCME using full-resolution HCP datasets and compare the results to other methods. Detailed schematics, layouts, source code, and logic implementation are provided in the appendix of this report.

1.1 Research Aims

In an attempt to map the connectivity of the human brain, a group of researchers led by the University of Minnesota, Washington University, and Oxford University focused on the neurological mapping of 1,200 healthy adults, in what is known as the NIH funded Human Connectome Project (HCP) [4]. As mentioned before, an issue with defining connectivity maps is in determining the appropriate resolution for the underlying correlations. To do this requires establishing connectedness for 10^9 to 10^{12} links, which has a high computational cost [5, 14, 15, 16]. For instance, computing a voxel-level Pearson correlation for one HCP scan on a multi-core CPU operating would take approximately 120 hours [5, 17, 14]. The enormity of the processing is seen considering in the Washington University / the University of Minnesota HCP dataset, that there are over 5,000 subjects (over 100 TB of datasets), with rs-MRI, fMRI (with seven tasks), DWI, and various other scans. The proposed HPCME system will enable the rapid correlation processing of a single participant's dataset in the HCP in a matter of hours and not days.

By reducing the processing time, neuroscience researchers will be able to understand the various statistical links better so they may be able to determine the effect of aging in the ongoing Lifespan HCP where both development and aging subjects are being scanned [2]. When combined with the broader goal of being able to correlate this data with subjects that have neurological diseases or mental disorders, the need for a method to rapidly correlate the rs-MRI and fMRI scans becomes clear.

Studies have shown that key local information is revealed when examining voxel-wise intrinsic correlation contrast (ICC) as opposed to ROI-based correlations [18]. ICC relies on SVD data reduction, and it remains to be seen if additional local features may be revealed when performing full-resolution voxel-wise correlation.

The innovation and central focus of this research is both in the particular hardware architecture and the integrated processing algorithm. The HPCME system enables rapid data-driven correlation of the HCP datasets by utilizing a modular design, multiple FPGAs, and data organization and a streaming architecture. The HPCME platform gives a researcher the ability to use a dedicated

hardware platform to compute in both a parallel and pipelined fashion, enabling the use of different signal processing algorithms. As designed, the HPCME calculates the Pearson Product Correlation Coefficient (PPCC) and performs the Seed-Based Correlation Analysis (VVCA) on HCP datasets to determine connectedness. Other algorithms, such as PCA (Principal Component Analysis), ICA (Independent Component Analysis), or for DTI applications using BEDPOST, could be added, in future revisions, with only the modification of the underlying structure in the FPGA. The HPCME system provides a researcher a stand-alone platform for analysis of large datasets. The aims of this research are as follows.

1.1.1 Proof-of-Concept HPCME Hardware Platform

The HPCME utilized an FPGA to process a synthetic HCP dataset using VVCA. The initial specifications, system architecture, hardware, communications, test vectors, and coding was completed for one system.

The test dataset that was used to perform the proof-of-concept analysis was based on a synthetic fMRI data set that had varying numbers of voxels and length. The number of voxels ranged from 20,000 to 1,000,000 voxels with a sample size of 284 to 2,048 samples. The test results were compared for accuracy against results generated using a MATLAB script with the same input dataset. The timing results were also compared to that of those generated by the recently published GPU-PCC algorithm [19, 20, 21], thereby comparing the performance of a GPU solution to this FPGA based solution.

1.1.2 HPCME System

With the proof-of-concept HPCME system operating at the expected performance targets set in Aim 1, a modular system, with four modules was arranged to increase the computation to meet the overall performance objectives of this research. A workstation application was also developed to monitor and control the various modules.

To test this modular system two selected HCP datasets, subjects 100307 and 102816, were used. These datasets utilized 902,629 voxels with a sample size of 284 and 1,200 samples respectively.

Similar to Aim 1, the results were also be compared to that of those generated by the GPU-PCC algorithm, or similar size, thereby again comparing the performance of a GPU solution to this modular FPGA based solution.

1.1.3 Validation of HPCME System - HCP Processing and Analysis

The HPCME system developed in Aim 2 was tested and validated using various datasets from the HCP and compared to correlations and analyzes performed via the CONN toolbox on the Brazos Cluster at Texas A&M University. Dr. Orr, an assistant professor at Texas A&M University, specializing in the area of cognition and cognitive neuroscience, compared the results between the two methods. The processing times of the HPCME was also compared to results generated by other methods, such as GPU-PCC, and CPU algorithms. The results of these comparisons are shown in the validation section of this dissertation.

1.2 fMRI, rs-MRI, and the Human Connectome Project

MRI have become a powerful, non-invasive, medical imaging tool. An MRI scanner can produce a 3D image from the interaction of changing magnetic gradients and electromagnetic fields with the hydrogen nuclei which are common in the human body. The MRI scanner detects this interaction and produces a 3D set of data to form an image of the tissue scanned. Thus, MRIs can be used to create images that can view, among others, tumors, blood vessels, and internal organs [22].

The fMRI and rs-fMRI use the BOLD (Blood Oxygenation Level Dependent) signal to measure activity in the brain. BOLD is the theory that in neurological areas of the brain that are active, there is the need for more oxygenated blood replacing the deoxygenated blood with a small-time lag after the start of the activity, which is called the hemodynamic response (HR). Oxygenated hemoglobin (Hb), which is paramagnetic, and deoxygenated hemoglobin (dHb), which is diamagnetic, can be differentiated in their responses to the MR scans [22, 23].

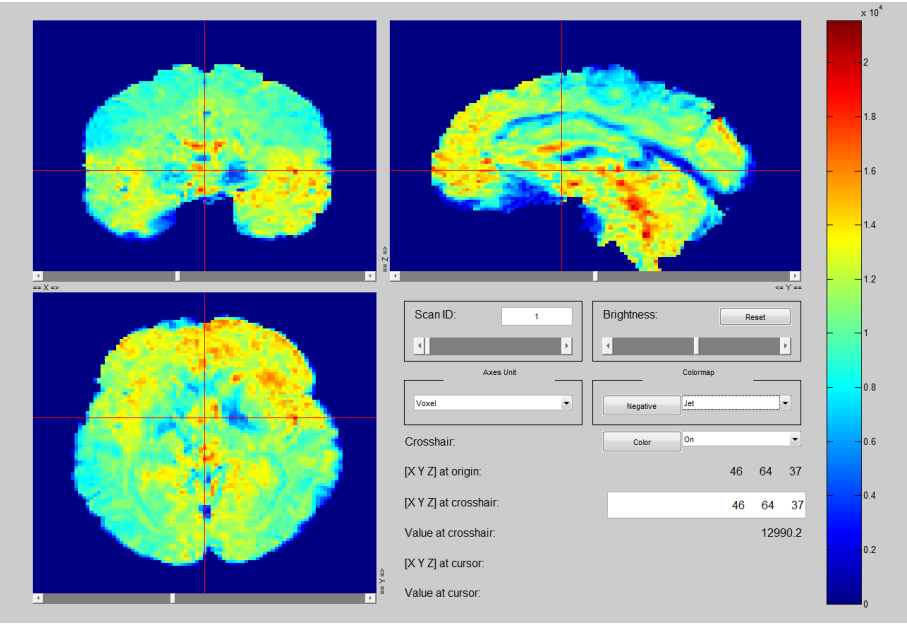


Figure 1.1: fMRI HCP dataset from test subject #100307

The Human Connectome Project, as previously discussed, has a plethora of public data available for analysis. For instance, for the original young adult study alone, there is a pre-processed set of data for 1,200 young adults (ages 22-35) that includes the following: T1 weighted, T2 weighted, rs-fMRI, fMRI, DWI, along with associated behavioral data. Moreover, there are ongoing projects that will collect similar high-resolution data sets from almost 5,000 individuals ranging in age from 0-100+ [2, 24]. There are also datasets currently being collected from about 4,000 individuals with human diseases, such as epilepsy, sporadic frontotemporal degeneration, and Alzheimer's [25, 26].

The rs-fMRI and fMRI HCP data are 4D data sets which include a group of 3D MRI spatial encoding sequences, including gradient echo and echo planar (EPI). Participants in the HCP study provided written consent and were scanned as per procedures outlined in the IRB (#201204036). Minimal preprocessing was done on the collected data to correct for field-map processing, rigid-body realignment, gradient distortion, and brain masking.

The typical resolution of the HCP datasets under analysis in this research is 91 voxels in the coronal plane, 109 voxels in the sagittal plane, and 91 voxels in the transverse plane. The sample depth ranges from about 284 to around 1,200 samples. Given this, the data sets typically have 902,629 voxels with a total of 1,083,154,800 sample data points. It should be noted that of the 902,629 voxel there can be a range of 72% to 99% that contain brain tissue.

In fMRI, it is common to correlate the time courses by using a "seed voxel" with the remaining voxels. However, this approach can be biased depending upon where the "seed voxel" is positioned, as bias can be introduced. For example, connectivity studies that are based on coarse brain atlases, with less than 100 regions, and more recent ones using finer parcellations, may bias the results by making anatomical assumptions [14]. However, by correlating all of the time courses this inherent bias is removed, but at the cost of much-increased computation time.

1.3 Voxel-to-Voxel Correlation Analysis (VVCA) and Connectedness

One typical method to generate brain connectivity maps is by using voxel-to-voxel correlation by Voxel-to-Voxel Correlation Analysis (VVCA). This type of analysis uses the Pearson Product Moment Correlation Coefficient (PPMCC) to provide a metric on the correlation between two independent variables [17]. Various researchers support that the Pearson correlation coefficient appears to be a good initial basis for understanding the correlation between pairs of voxels [14, 9, 27, 12]. The Pearson correlation coefficient will have a value of +1.0 for total positive correlation, -1.0 for total negative correlation, and a zero for no correlation between samples. The Pearson Correlation Coefficient (PCC) between two N-sample time-series, P and Q, is defined as:

$$r_{(P,Q)} = \frac{Cov(P,Q)}{\sigma_p \sigma_q} = \frac{E[(P-\mu_p)(Q-\mu_q)]}{\sigma_p \sigma_q} \tag{1.1}$$

For the Pearson correlation coefficient of a sample set of P and Q defined and with N being the

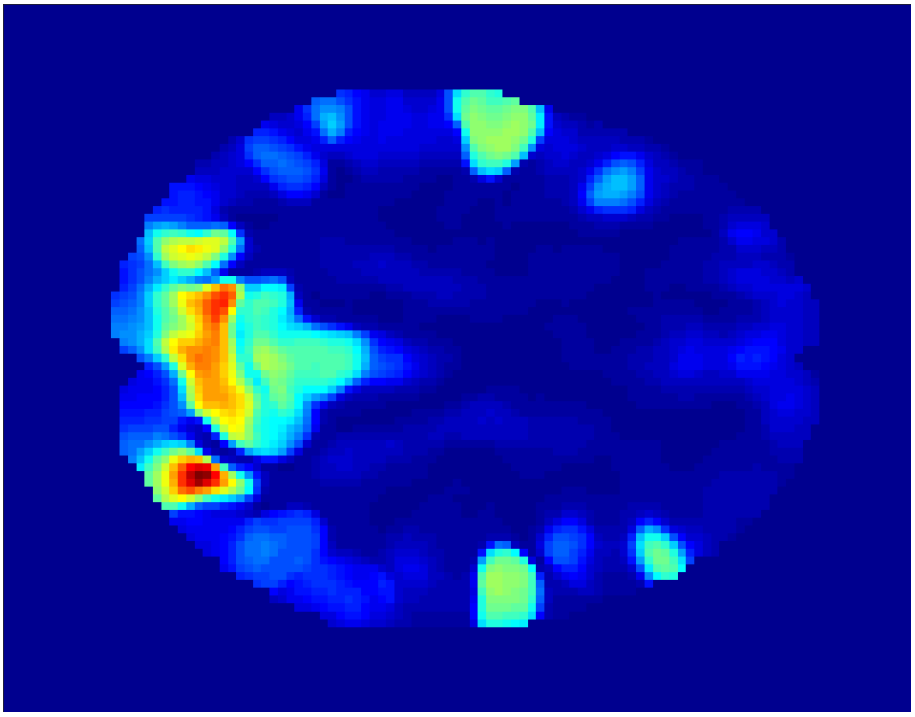


Figure 1.2: Example of node degree distribution for 0.63% completeness

number of samples in a data set:

$$r_{(P,Q)} = \frac{\sum_{i=1}^{N}(P_i - \mu_p)(Q_i - \mu_q)}{\sqrt{\sum_{i=1}^{N}(P_i - \mu_p)^2}\sqrt{\sum_{i=1}^{N}(Q_i - \mu_q)^2}} \tag{1.2}$$

A "seed voxel," or the reference, is then correlated to the remaining valid voxels in the fMRI time series. In this research, the variable P is to be the "seed" voxel, which is chosen from the selected subregion, and then the variable Q is all remaining voxels in the sample data set. The number of correlation coefficient calculations to process every voxel to the other voxels, i.e. to determine the whole brain correlation relationship requires approximately 5.86 x 10^{17} operations. As can be seen, to perform the number of correlation coefficient calculations this can become quite extensive even for a relatively small volume.

To determine if a neurological region connects to another area we consider if the correlation between sample pairs, i.e. voxel pairs, exceeds a specified threshold value (r'). Connectivity for a given pair can be defined by:

$$e_{(P,Q)} = \begin{cases} 1 & \text{if } r(P,Q) > r' \\ 0 & \text{otherwise} \end{cases} \tag{1.3}$$

Therefore, for each voxel pair, there is a binary state of "connection" or "no connection". For a "seed" voxel this will represent all voxels that can connect to itself.

Node Degree is based upon the total number of connections for a particular voxel, or node, and P is simply defined as:

$$k_P = \sum_{i=1}^{M} e_{P,i} \tag{1.4}$$

Completeness is a metric that represents the value of connections for a particular voxel versus the maximum number of total observed connections of the whole brain.

$$Completeness = \frac{k_P}{MaxConnections} \tag{1.5}$$

1.4 CONN: Functional Connectivity Toolbox

The CONN functional connectivity toolbox implements various statistical analysis tools to aid in the identification of connectivity networks from fMRI and rs-fMRI datasets. CONN is a cross-platform toolbox that is Matlab/SPM based and supports various connectivity and statistical processing methods, including graph theory methods, ICA, ROI to ROI (atlas based) functional correlation, seed-based correlations, and component-based noise correction method (CompCor) among others. CONN enabling both preprocessing, computation, analysis, and visualization of the connectivity data from these databases [28, 29].

Processing and analysis steps include [29]:

- Importing ANALYZE, DICOM, and NIfTI files
- Structural and functional segmentation
- Functional artifact rejection
- Functional smoothing
- Regression-based denoising
- Statistical quality control methods
- Connectivity Analyses, including Seed-Based Correlations (VVCA), Complex Network Analyses, and ICA.
- Comparison methods and models

The CONN toolbox can be implemented with distributed clusters, as has been done with Dr. Orr's research and the Brazos Cluster, or in a multicore desktop environment and automatically parallelize processing steps. The toolbox is controlled via a user-friendly interface, shown in Figure 1.3.

CONN has been cited in over 1,000 articles and has over 2,000 registered users of the toolbox [29]. Because of the wide acceptance in the neuroscience community, the CONN toolbox is being used as a point of validation for the HPCME results based upon the same HCP datasets.

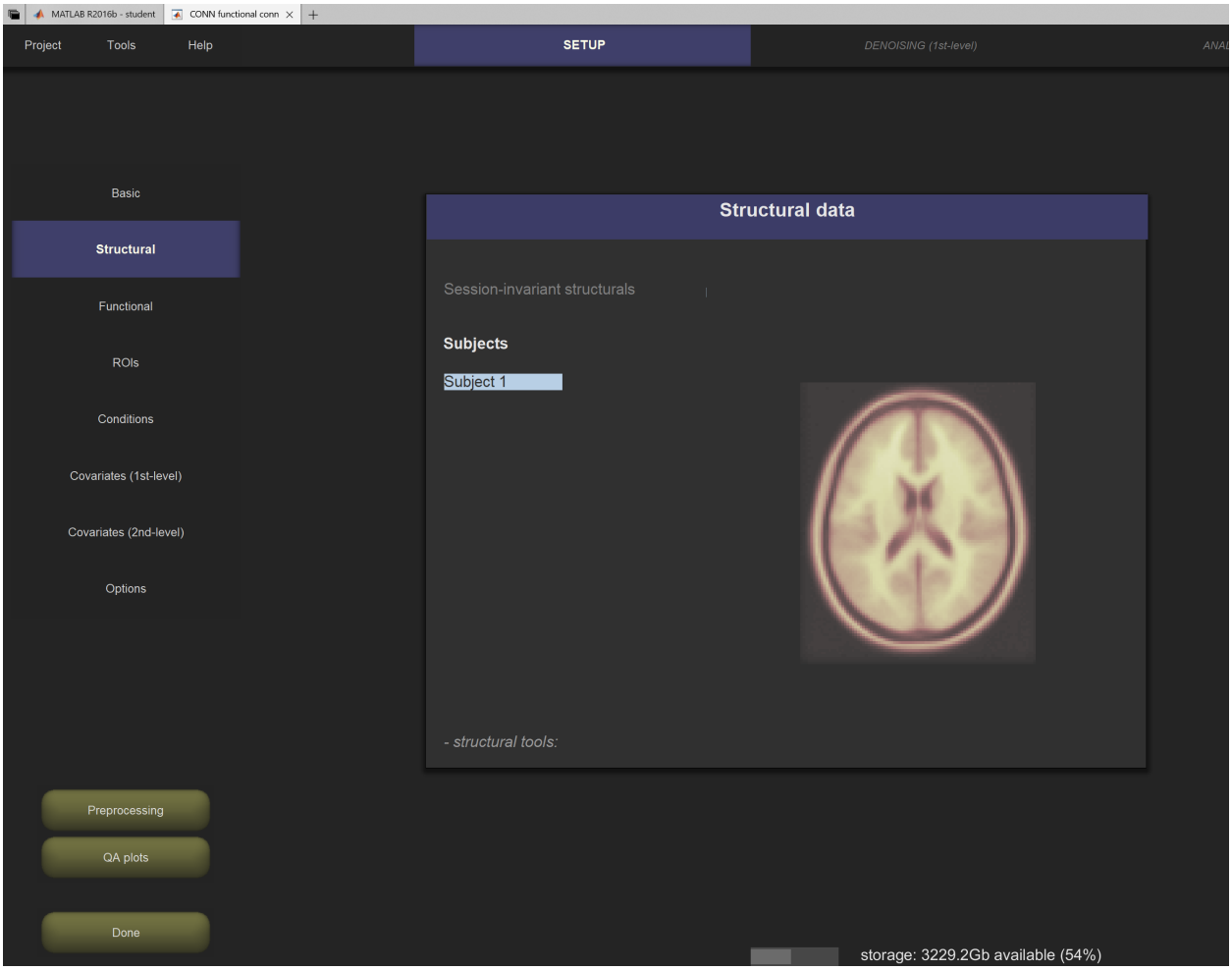


Figure 1.3: CONN Toolbox Application Interface

1.5 GPU-PCC Technique

As discussed in the previous sections, Pearson's Correlation Coefficient (PCC) is a useful means for determining functional connectivity. However, as previously discussed, high-fidelity, whole-brain, and voxel-to-voxel based PCC is computationally burdensome. One recent algorithm for improving processing of PCC of big fMRI datasets is titled GPU-PCC [19, 20] and is based on using a GPU (Graphics Producing Unit) to speed up processing.

A Graphics Processing Unit (GPU) are a group of coprocessors that initially were designed to produce high-quality and rapid graphical renderings in video games, such as more realistic 3D renderings. Over the past several years GPU's, due to their many streaming processing cores and memory architecture, have been leveraged to process algorithms that are computationally expensive. For instance, the NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library can perform matrix and vector-based multiplications efficiently.

The GPU-PCC method, is a GPU based method which is based upon the use of a vector dot product to compute Pearson's Correlation Coefficient for each pair in the dataset. This method was evaluated, by Eslami et al., by using synthetic and a real small-subset HCP fMRI dataset with a traditional CPU implementation (developed in C++), and with an existing GPU method, General Matrix-Matrix Multiplication (GEMM) [19].

The developers claim that the GPU-PCC algorithm is 94.62 times faster than the traditional CPU methods and 4.28 times faster than the existing GPU based techniques on a fMRI dataset with up to 90,000 voxels [19].

A comparison of similar voxels and time course sample sizes was performed on their implemented code. These results are discussed in the validation section. The GPU-PCC source code used for this comparison is publicly available as a GPL license on a GitHub portal of their lab at https://github.com/pcdslab/GPU-PCC.

2. METHODOLOGY

2.1 Architecture of HPCME

The HPCME was designed to utilize a modular system comprised of various independent blocks calculating their assigned seed group of the same pre-processed, and organized voxel data sets. Each modular block has a parallel computation core and an independent and dedicated memory system to enable the near continuous streaming of the voxels and seeds being processed using the previously discussed VVCA method. The overarching concept was to break down the numerous operations into optimally sized blocks.

Mathematically most of the computation cycles in VVCA are for performing multiplications with accumulation, and illustrated in the following equation for the numerator of the correlation coefficient:

$$\sum_{i=1}^{N}(P_i - \mu_p)(Q_i - \mu_q) \tag{2.1}$$

Though this equation on the surface is simple, and would conceptually be easy to implement on any computer workstation, it is, in fact, difficult, not due to the complexity, but instead, because it repeats for each pair of voxels. In MATLAB for instance, a correlation matrix for a HCP dataset with dimensions of $91 \times 109 \times 91$ is 6.517×10^{12} bytes (5.92 TB).

Various FPGA devices serve as the central processing core to achieve the goal of performing these VVCA calculations quickly. FPGAs have the advantage of being customized into various digital logical systems by changing their underlying logical gate structure. In the HPCME, the FPGAs enabled the ability to perform both parallel and pipelined operations.

With parallel processing, many of the same mathematical operations process at the same time, just like in a GPU or a multi-core CPU. In other words, if one has a process that is independent of another, it can be calculated simultaneously. Such is the case in VVCA, as many seeds are processing against the same voxel independently.

Pipelining is where, for each clock tick of the FPGA, the processing is performed for a step

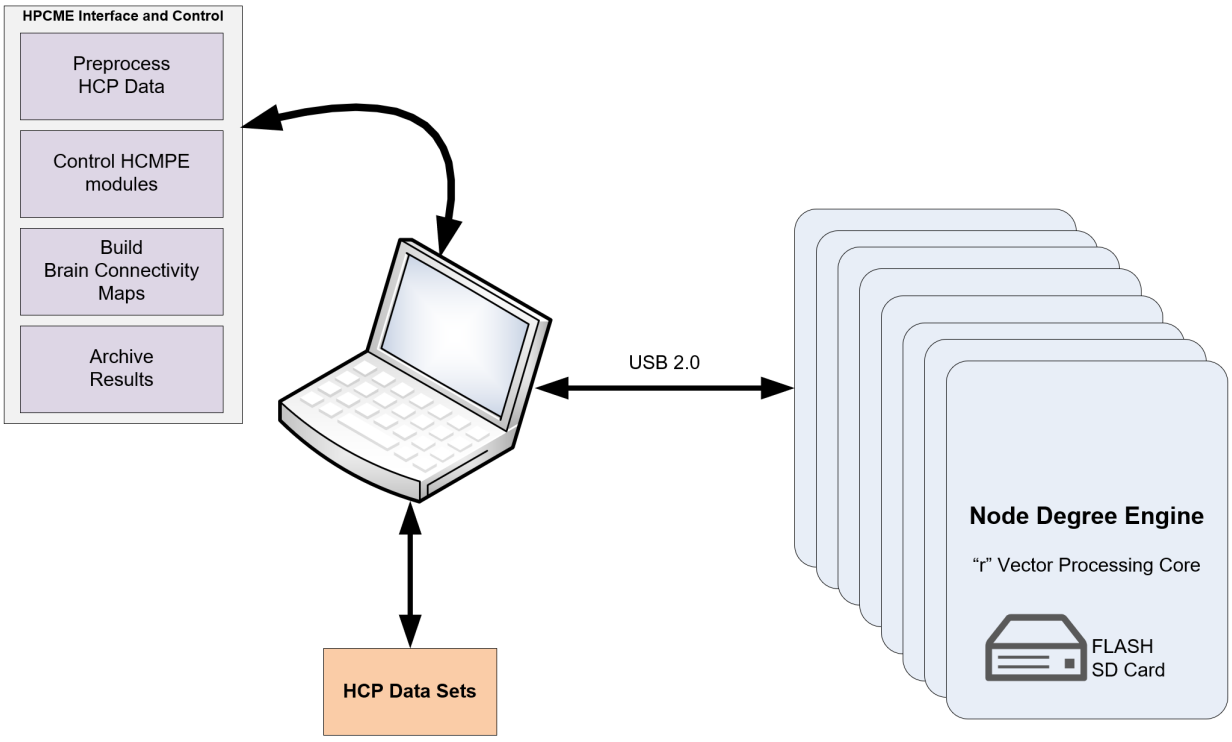


Figure 2.1: Fundamental HPCME Architecture
https://www.sciencedirect.com/science/article/pii/S1877750317310633

while the previous step can now be loaded and processed with new data. If the latency of the operation (i.e., the time it takes to finish) is not as significant as the need to handle new data continuously, this can be very useful. The tradeoff is in the use of more fabric resources. In this type of operation, steps are spaced in time by clock-ticks and limited only by the ability of the FPGA fabric to meet timing and have sufficient resources.

Figure 2.1 shows the overall architecture of the HPCME system and Figure 2.2 shows the data flow from the initial HCP dataset to the final results being analyzed by a researcher. The HPCME utilizes various FPGA systems connected to a PC workstation. The PC workstation pre-processes, controls, monitors, and generates the final graphical brain connectivity maps from the FPGA subsystem. The FPGA subsystem, known as the Node Degree Engine (NDE), was designed to process the voxels and seed data and returns the resulting r vectors to the PC workstation for postprocessing.

The HPCME currently communicates to the several NDEs in the system via USB 2.0, for monitor and control, and FLASH SD cards for data file processing. The reasoning for using FLASH SD cards for this implementation was due to the data transfer rate and existing file system support in both the Xilinx Zynq platform as well as support for most existing PC workstations.

The HPCME is designed to not require monitoring or control during operation. In fact, other than loading in the SD cards with data and starting the initial process, there is little to no required interaction with the HPCME, thus enabling the researcher to focus on the postprocessing of results while new data is processing.

2.2 Architecture of Node Degree Engine

The Node Degree Engine (NDE) is a crucial component in the HPCME system. The NDE processes the HCP dataset and produces a resulting set of connected voxel pairs. Each NDE incorporates power, communications, and memory infrastructure that enables it to be independent of the PC workstation and the other NDEs in the HPCME system. The NDE has configuration settings to allow it to know which part of the voxel map to process, which seeds to use, and what

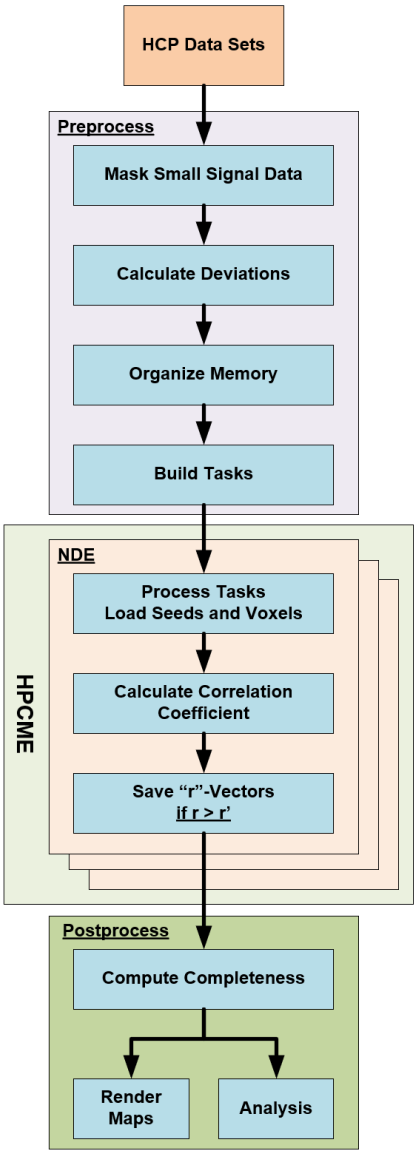


Figure 2.2: HPCME Processing Data Flow

the minimum correlation coefficient threshold for a connection should be.

The NDE is a hybrid FPGA system that has an embedded dual-core ARM processor. In this configuration, one of the CPUs is assigned to merely move the seeds and voxel datasets from the non-volatile storage medium (i.e., FLASH SD card) into the high-speed DDR3 SDRAM memory based upon the assigned streaming task list. Also, based on the assigned streaming task list, a group of seeds is transferred from the non-volatile storage medium into SDRAM for subsequent transferring to Block RAM (BRAM). A preferred architecture is a system with a significant amount of SDRAM, as more voxel data may be stored in SDRAM versus being fetched continuously from the external FLASH memory, such as an SD card.

A DMA is utilized to stream the voxel data from the SDRAM to the correlation coefficient processing core. This processing core also connects to the seed BRAM via a large width data bus. It is important to note that the voxel inputs are of a streaming configuration, whereas the seed data is a BRAM interface. This architecture is shown in Figure 2.3.

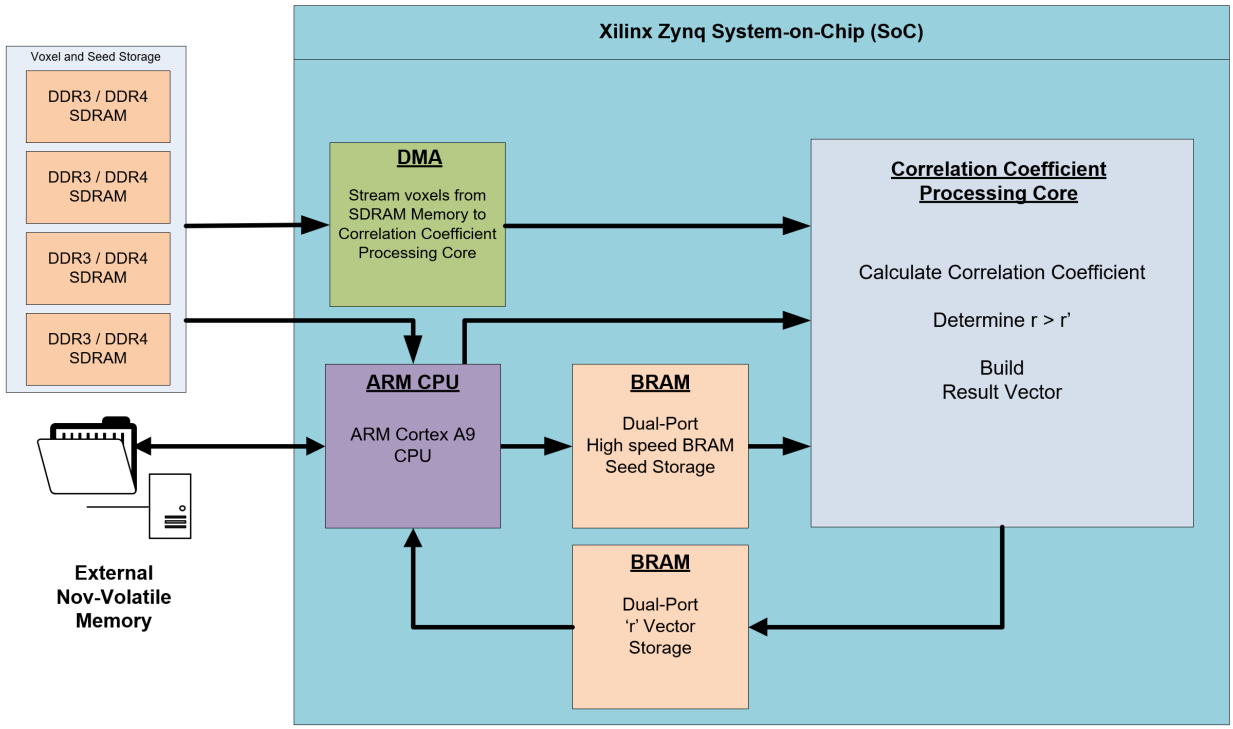


Figure 2.3: NDE SoC FPGA General Architecture

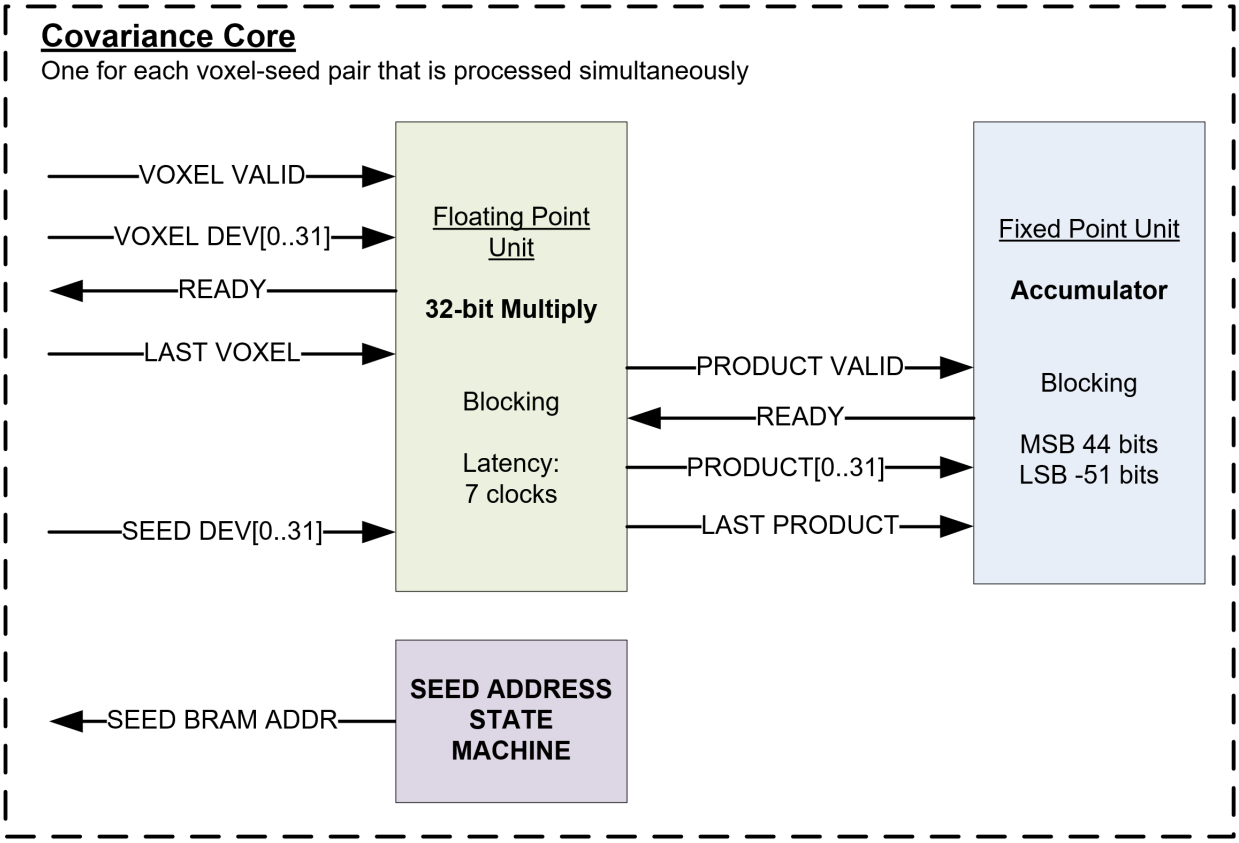


Figure 2.4: NDE Multiplication - Covariance Computation Block Diagram

For each voxel and seed pair, there is a multiplier and an accumulator. The multiplier is a single precision 32-bit floating point mathematics core that takes the seed and voxel data value and produces the product of the two. A logical block diagram of this is shown in Figure 2.4. The floating point cores are made to block until the operands are valid and the results data path is ready, as illustrated by the handshaking logic shown in the diagram. For Xilinx based FPGA design, utilizing the Zynq SoC, all data handshaking is done with reference to the AXI streaming standard referenced in the Xilinx AXI Reference Guide [30].

For this specific implementation, the latency of the multiplier was seven clock cycles, as to reduce logic resources used. A state machine produces the correct seed BRAM address (i.e., seed count), which is illustrated in Figure 2.5. Immediately after this multiplication is an accumulation of the "N" samples, which the value of "N" is set for the core during setup and configuration, and

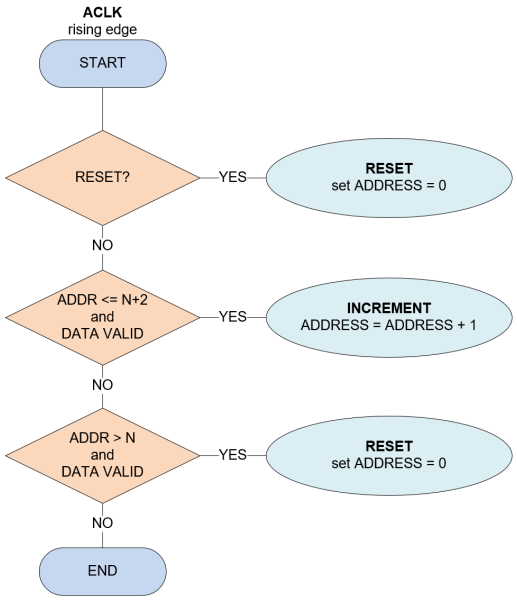


Figure 2.5: Seed BRAM Address Generator - State Machine

corresponds to the equation of:

$$\sum_{i=1}^{N}(P_i - \mu_p)(Q_i - \mu_q) \tag{2.2}$$

The accumulator IP core has as inputs and outputs of single precision floating point values, but internally it uses fixed-point mathematics to reduce the necessary fabric resources. For this particular instance, the accumulator core has an MSB of 44 bits and an LSB of -51 bits. The core has a maximum MSB input of 30 bits, which is more than sufficient for the range of data that is expected between the voxel and the seed deviations. As such, the total width of each accumulator is 96 bits. For this specific implementation, the latency of the accumulator was 27 clock cycles, as to reduce logic resources used.

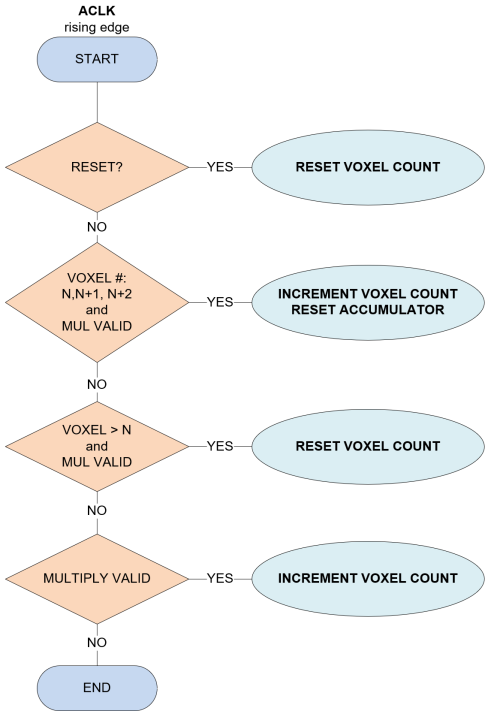


Figure 2.6: Accumulator Reset - State Machine

The accumulator resets to zero upon the start of the core and after N+1 and N+2 samples occur, which is illustrated in Figure 2.6. The value of the multiplication between the seed standard deviation and the voxel standard deviation is stored in an independent register at N+1 samples, and at N+2 samples the voxel index is stored. Another register stores the accumulation at N samples. A logical block diagram of this is shown in Figure 2.7.

After this process, and while new voxel and seed data are streamed in for the new accumulation values, the previous accumulation values including the previous index and standard deviation multiplication are used to determine the Pearson Correlation Coefficient (i.e., PCC r value), of which the equation is as follows.

$$r_{(P,Q)} = \frac{\sum_{i=1}^{N}(P_i - \mu_p)(Q_i - \mu_q)}{\sqrt{\sum_{i=1}^{N}(P_i - \mu_p)^2}\sqrt{\sum_{i=1}^{N}(Q_i - \mu_q)^2}} \tag{2.3}$$

Processing the r value through a multiplexed state machine reduces the number of logical

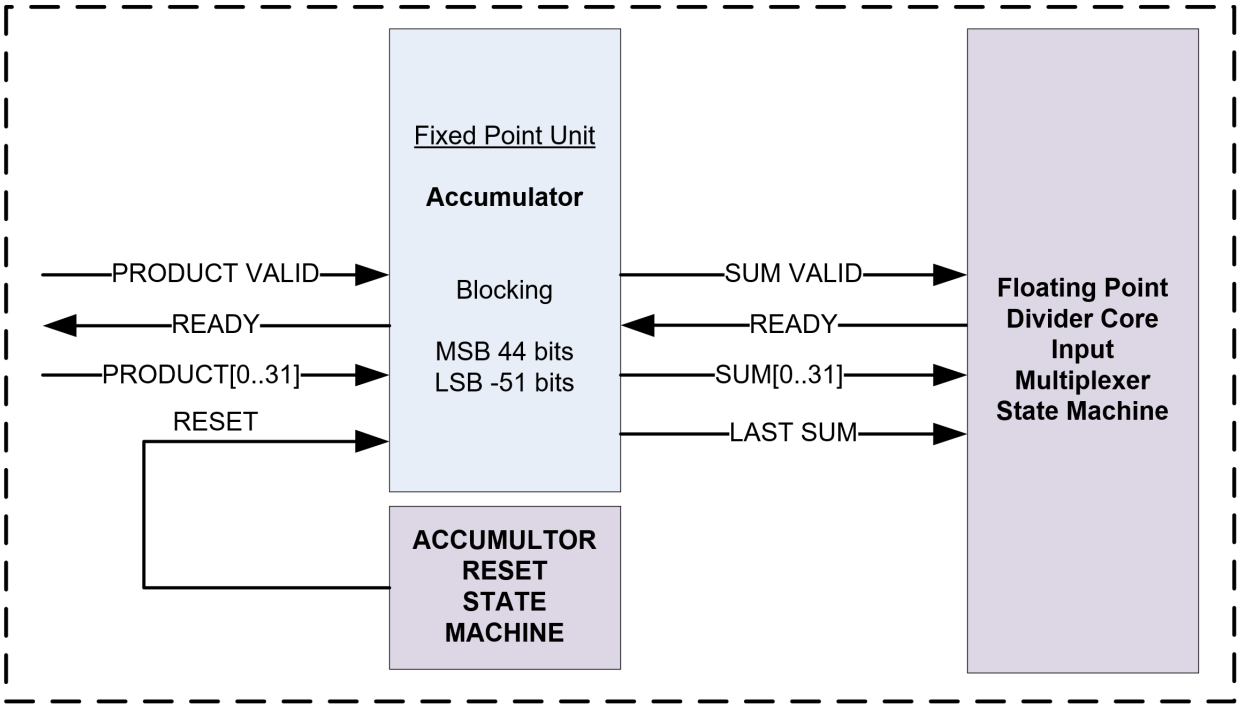


Figure 2.7: NDE Accumulator - Covariance Computation Block Diagram

resources consumed. There are a group of floating point dividers and comparators that each processing a set number of paired data including the numerator (i.e., covariance) and the denominator (i.e., standard deviations). The state machine processes through one group at the time for a total of eight r pairs. It is crucial to have sufficient blocks of resources to enable the processing of all of the voxel-seed correlation pairs within 128 clock cycles or less. This threshold enables the core to consistently stream new data all while generating correlation results as long as there are at least 128 samples per voxel time-series dataset. In other words, the N sample size of the data set must be higher than the total correlation processing latency.

In this core, the r value, which is the result of the divider operation, is compared against the r' threshold value using a floating point comparator. Setting r' is accomplished at the start of the core during task setup configuration. Another floating point comparator is used to verify that the value is less than or equal to 1.0, recognizing that this is a proper Pearson Correlation Coefficient, as is necessary when there can be standard deviation values that include zeros, such is the case with

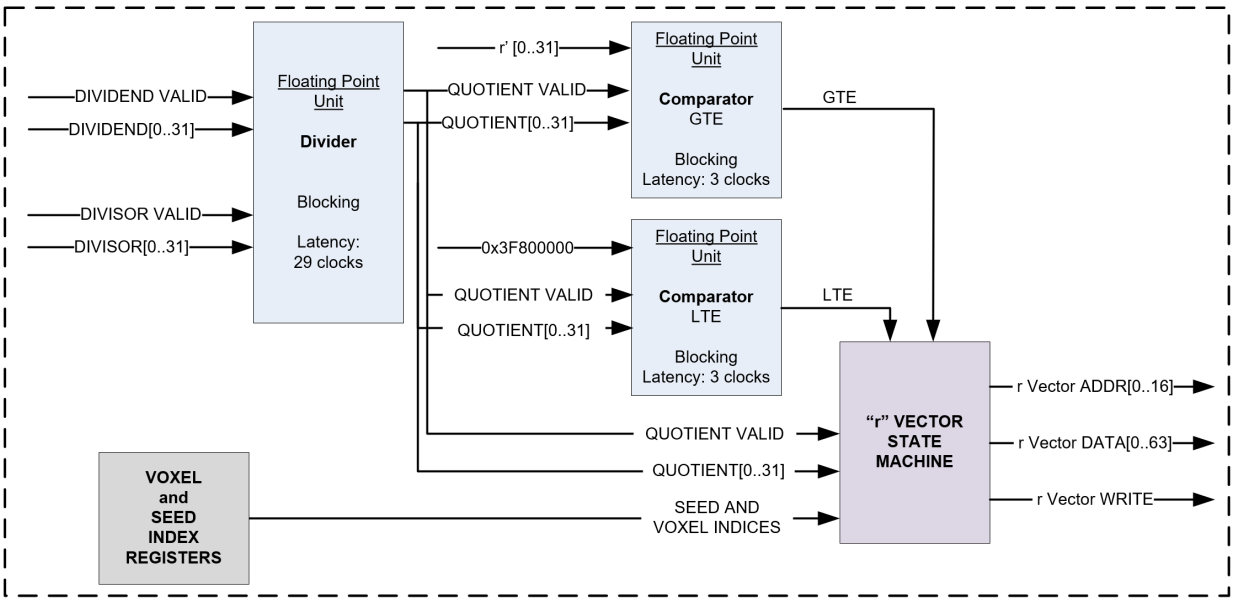


Figure 2.8: NDE Correlation Coefficient Calculation and r Vector Generation

"null" data. These $null$ values will compute an r value of infinity and thus will be ignored. This "null" capability can be used for masking data, as is the case with preprocessing data.

When an r value is GTE r' but LTE to 1.0 then the value including the index is stored into the next available address result, in the result BRAM. This value represents a result vector as it contains the voxel index, the seed index for the block of seeds being processed, and the r value. These cores and process are shown in Figure 2.8.

Decoding the actual r vector is accomplished by realizing which seed group was being processed when the result vector was created. Knowing the seed group and the seed index one can regenerate the entire r vector with minimal computational effort, which is illustrated in the source code, shown in Figure 2.9, to read and process results.

The current address of the result memory is provided as an output of the core so the CPU that is controlling the data streaming can monitor whether or not to pull the r vectors from the BRAM and store them in permanent nonvolatile FLASH memory, such as an SD card or SSD.

In the end, there are Ns by Nv processing pairs, with Ns being the number of seeds and Nv being the number of voxels processed simultaneously. These processing pairs determine the co-

```
// Process r-Vector Files
if (File.Exists(fileName) == true)
{
    // Open a file stream to the file and binary reader
    FileStream fs = File.Open(fileName, FileMode.Open);
    using (BinaryReader br = new BinaryReader(fs))
    {
        // Process until the EOF
        while (fs.Position < fs.Length)
        {
            // Read results header
            int iVoxelGroup = br.ReadInt32();
            int iSeedGroup = br.ReadInt32();
            int iOffset = br.ReadInt32();
            int iNumBytes = br.ReadInt32();
            int iNumResults = iNumBytes / 8;

            // Read data, make sure mod of 8 bytes (i.e. result vector packet is 64 bits)
            if ((iNumBytes % 8) == 0)
            {
                // Step through all the results
                for (int iRes = 0; iRes < iNumResults; iRes++)
                {
                    // Read "r" value and index
                    // Read floating-point value (Correlation Coefficient)
                    float rValue = br.ReadSingle();

                    // Read the 32-bit Index Value
                    int rIndex = br.ReadInt32();

                    // Get the Voxel Index for this Coefficient - (upper 27 bits)
                    int iVoxelIndex = rIndex >> 5;

                    // Get the Seed Index, based on seed group in header,
                    // number of seeds, and stored index
                    int iSeedIndex = (iSeedGroup * iNumSeeds) + ((rIndex >> 1) & 0xF);

                    // Increment connectedness map by the indices
                    voxelmap[iVoxelIndex]++;
                    voxelmap[iSeedIndex]++;

                    // Do other analysis as desired...
                }
            }
        }
    }
}
```

Figure 2.9: r-Vector Results Processing (C#)

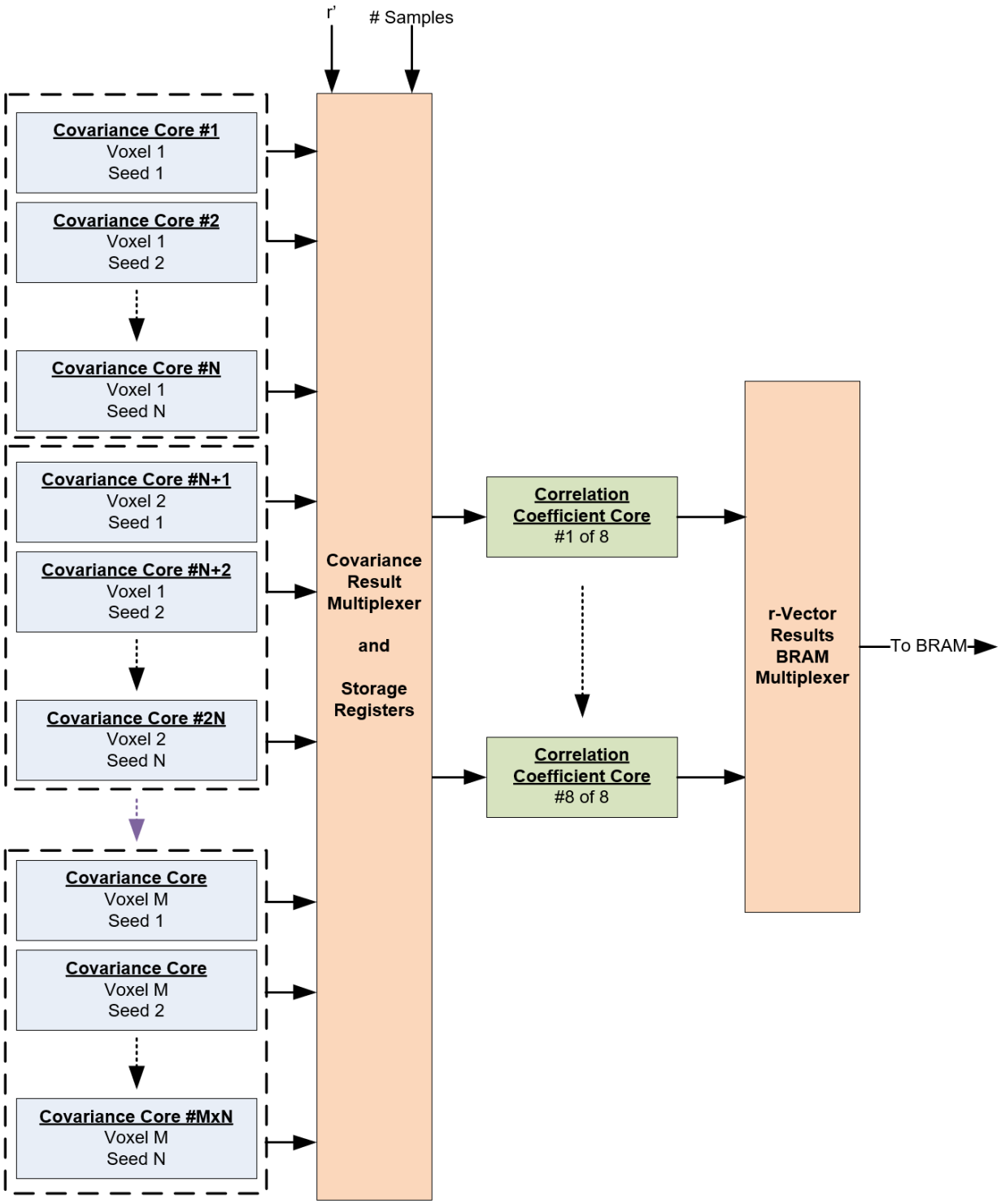


Figure 2.10: Layout of Logic for NDE System Architecture

variance and a set divider/comparator groups that process the final r value representing all the floating-point operational blocks being utilized for this architecture. The number of seed-voxel pair processing is entirely dependent upon the logical resources available for a given FPGA or ASIC. For the system implemented in this research, this value was either 32 or 64 seed-voxel pairs. The overall system architecture with the logical components previously described is shown in Figure 2.10.

The NDE enables the near constant streaming of data as long as the data stream has embedded metadata included with it, such as the voxel and seed indices and the standard deviation values. Given this logic arrangement, there is no need to pause the processing to fetch variables as they can be sequentially and continuously processed.

To get the best performance out of the NDE, it is crucial to have a core design that has a wide voxel port to enable efficient streaming from the DDR3/4 SDRAM. For instance, a Xilinx DDR4 based MIG can operate at more than 17 GB/sec. Another critical architecture design element is having a sizeable dual-port BRAM interface for both simultaneous access to both the seeds and the resulting r vector.

The overall NDE architecture enables efficient processing of HCP datasets by using state machines, various parallel floating point cores, and independent dual-port BRAM. This architecture, when organized as a group of independent NDEs enables full processing resolution and deep sample depth HCP datasets in a matter of several hours.

2.3 Preprocessing of HCP Data

HCP dataset must first be preprocessed for the NDE to process efficiently. The HCP dataset, which is in the NifTI (Neuroimaging Informatics Technology Initiative) file format [31], is preprocessed before being computed by the NDE. This preprocessing includes calculating the deviations from the mean, the standard deviation of all voxels, determining the correlation of the seed/voxel diagonal of the correlation matrix, and dividing the processing workload to each NDE in the system. The voxels and seeds are stored in a format for efficient processing and streaming from the local memory into the processing core. This memory organization format for the voxels and seeds is shown in Figure 2.19 and Figure 2.20 in the following section, and the steps for the preprocessing are shown in Figure 2.11.

STEP 1 **Load HCP Data File (NeuroImaging Informatics Technology Format)**
- Get number of voxels
- Get number of samples

STEP 2 **Flatten HCP dataset into 2D array (Voxels with Samples)**

STEP 3 **Mask "low-level" signals**
- Find data above arbitrary threshold and set a mask bit to 1.0
- Perform a dot-product multiplication of mask and data set

STEP 4 **Compute voxel statistics**
- Compute voxel sample mean
- Compute voxel deviations
- Compute voxel standard deviation

STEP 5 **Pad voxel data**
- Make sure voxel array is of the correct dimensions to process
- the number of voxels and seeds simultaneously

STEP 6 **Build voxel data files**
- Group data by the number of voxels processed in parallel (i.e., 2 voxels)
- Store voxel deviation data as per memory organization (N samples)
- Store the standard deviation of the voxels in this group
- Store the indices of the voxels in group
- Repeat until end of voxel array

STEP 7 **Build seed data files**
- Group data by the number of seeds processed in parallel (i.e., 16 seeds)
- Store seed (i.e., voxel) deviation data as per memory organization (N samples)
- Store the standard deviation of the seeds in this group
- Store the indices of the seeds in group
- Repeat until end of voxel array

STEP 8 **Build task files**
- Group data by number of NDE systems
- Build task listing each seed and voxel pair group to be processed
- Save task list to task file for processing on NDE

Figure 2.11: Steps for Preprocessing HCP datasets for use on the HPCME and NDEs

In the current architecture of the NDE, there is a streaming core that correlates multiple voxels simultaneously with a group of seeds. For this to be accomplished the data must first be organized in such a way that we have the voxels, with all the samples coinciding along with their embedded standard deviation values and an embedded index. The integrated index enables the easy and rapid creation of the final r vector in the final processing of the core.

Seed preprocessing is accomplished in a similar fashion where we store a group of seeds, along with the standard deviations, and the seed index. Though the index is included in the data stream, enabling data synchronization, the core ignores the index as it is not needed to generate the internal r vector.

As illustrated in Figure 2.11, the first step is to flatten the N cubes with M voxels each into a matrix of size M voxels (columns) by N samples (rows). Creating a mask extracts any low signal data (i.e., less than 200 counts), and then sets this data to zero. This mask is applied to the matrix we just flattened. Below is a MATLAB script of this operation.

Then the mean of each column is computed, and a deviation from the mean for each sample

```
%--------------------------------------------------------------------------
% Load the NeuroImaging Informatics Technology data file from
% the Human Connectome Project
% http://www.humanconnectome.org/
% Use library of functions from the NIfTI
nii = load_nii('119833_srfMRI_REST1_LR_hp2000_clean.nii');
NIfTIDataset = nii.img;

%--------------------------------------------------------------------------
% Get number of voxels and samples in the data file
NumVoxels = size(NIfTIDataset,1) * size(NIfTIDataset,2) * size(NIfTIDataset,3);
NumSamples = size(NIfTIDataset,4);

%--------------------------------------------------------------------------
% Flatten the volume to a 2D array (i.e. voxels and samples)
Voxels = reshape(NIfTIDataset,NumVoxels, NumSamples)';

% Build data mask to pull out "low signal" data
maskI = find((Voxels(:,:) > 200));
MaskObj = zeros(NumSamples, NumVoxels);
MaskObj(maskI) = 1.0;

% Apply to voxels as dot-product, mask "low signal" data
Voxels = Voxels .* MaskObj;
```

Figure 2.12: Preprocessing Step 1 through 3 (MATLAB Script)

point is computed, and this becomes the Voxel Deviation Matrix (VDM). Next, we compute the standard deviation of the voxel data which is multiplied by N. All of this, on a typical workstation, requires less than a minute of processing time. Below is a MATLAB script of these operations and the corresponding equation.

```
%-------------------------------------------------------------------------
% Compute the statistics on the data, determine the mean, deviations,
% and standard deviation
VoxelMean = mean(Voxels);
VoxelDeviations = bsxfun(@minus, Voxels, VoxelMean);
VoxelSDSum = std(Voxels) * sqrt(NumSamples - 1);
```

Figure 2.13: Preprocessing Step 4: Compute voxel statistics (MATLAB Script)

The data is then processed to make sure that it is of the correct dimension to be streamed through the engine, as the engine has specific seed and voxel widths that constrain the data input dimensions. For instance, since the NDEs compute data by Z voxel-seed pair operations simultaneously then the columns of the VDM must be a modulus of Z (i.e., the number of columns of the VDM mod Z must be zero). The MATLAB script of this operation is:

```
%-------------------------------------------------------------------------
% Verify that voxels are a mod of the number of seeds (i.e. 64) as we
% process two voxels and sixteen seeds at a time
if mod(NumVoxels, (NumSeeds * NumSystems)) ~= 0
    padSize =  (NumSeeds * NumSystems) - mod(NumVoxels, (NumSeeds * NumSystems));
    Pad = ones(NumSamples, padSize);
    PadSD = ones(1, padSize);
    VoxelDeviations = [VoxelDeviations Pad];
    VoxelSDSum = [VoxelSDSum PadSD];
end;
```

Figure 2.14: Preprocessing Step 5: Pad voxel data (MATLAB Script)

The VDM is then saved as a group of files for eventual use by the NDE, as both seed and voxels. In theory, if there were enough SDRAM on a single NDE it would not be necessary to split the VDM into separate groups. However, due to memory size limitation on the test system the VMD was split over various independent files that can be loaded into the SDRAM of the NDE.

Each implementation of the NDE will have different constraints and based on the sample depth a processing set may have only one group to as many as sixteen, as this is entirely dependent upon the available memory. The building of the voxel and seed data files are shown in the following scripts:

```
%--------------------------------------------------------------------------
% Set parameters  (i.e. 4 NDEs, 8 Groups, 16 Seeds, 2 Voxels)
NumSystems = 4;
NumGroups = 8;
NumSeeds = 16;
NumVoxelsProcessed = 2;


%--------------------------------------------------------------------------
% Export voxel files
% Now group the voxels into blocks
% i.e. 4 processing blocks, for memory constraint)
addr = 0;
for group = 1:NumGroups
    fprintf('Processing Group #%d of %d\n', group, NumGroups);

    NumVoxelsPerGroup = size(VoxelDeviations,2) / NumGroups;

    FileName = sprintf('voxel_%d.bin',group);
    VoxelfileID = fopen(FileName,'w');

    % Write information to file (voxel group #, # voxels to process, # samples)
    fwrite(VoxelfileID, group,'int');
    fwrite(VoxelfileID, size(VoxelDeviations,2),'int');
    fwrite(VoxelfileID, NumGroups,'int');
    fwrite(VoxelfileID, NumVoxelsPerGroup,'int');
    fwrite(VoxelfileID, NumSamples,'int');

    % Write # voxels to be processed simultaneously
    fwrite(VoxelfileID, NumVoxelsProcessed,'int');

    % Loop through all voxels in groups of 2, and output the sample data
    start_voxel = ((group - 1) * NumVoxelsPerGroup) + 1;
    end_voxel = (group * NumVoxelsPerGroup);
    fprintf('Processing Voxel #%d and %d\n', start_voxel, end_voxel);
    for VSi = start_voxel:NumVoxelsProcessed:end_voxel

        % Store deviation data of all samples for the two voxels
        for Sample = 1:NumSamples
            for Vi = 0:(NumVoxelsProcessed - 1)
                VoxelDevData = VoxelDeviations(Sample, VSi + Vi);
                fwrite(VoxelfileID,VoxelDevData,'float');
                addr = addr + 1;
            end;
        end;
        % Store standard deviation data of the two voxels
        for Vi = 0:(NumVoxelsProcessed - 1)
            VoxelSDData = VoxelSDSum(VSi + Vi);
            fwrite(VoxelfileID,VoxelSDData,'float');
            addr = addr + 1;
        end;

        % Store voxel #1
        fwrite(VoxelfileID,VSi,'int');
        addr = addr + 1;

        % Store voxel #2
        fwrite(VoxelfileID,VSi + 1,'int');
        addr = addr + 1;
    end;
```

Figure 2.15: Preprocessing Step 6: Build voxel data files (MATLAB Script)

```
%--------------------------------------------------------------------------
% Export seed files
% Now group the voxels into groups of 16 seeds
NumSeedsPerGroup = size(VoxelDeviations,2) / NumSeeds;

FileGroups = floor(NumSeedsPerGroup / (16*1024));
for SeedFile=0:FileGroups
    FileName = sprintf('seeds_%d.bin',SeedFile);
    SeedfileID = fopen(FileName,'w');

    % Write information to file (system #, seed group #, # seeds to process, # samples)
    fwrite(SeedfileID, NumSeedsPerGroup,'int');
    fwrite(SeedfileID, NumSamples,'int');

    % Write # seeds to be processed simultaneously
    fwrite(SeedfileID, NumSeeds,'int');

    groupstart = (SeedFile * (16*1024)) + 1;
    groupend = ((SeedFile + 1) * (16*1024));
    if (groupend > NumSeedsPerGroup)
        groupend = NumSeedsPerGroup;
    end;

    for group = groupstart:groupend
        fprintf('Processing Seed Group #%d of %d\n', group, groupend)

        % Determine voxel offset for system to process group
        % for a full set of data
        Offset = ((group - 1) * NumSeeds) + NumSeeds;
        fwrite(SeedfileID, group,'int');
        fwrite(SeedfileID, Offset,'int');


        % Loop through all voxels in groups of 2, and output the sample data
        start_voxel = ((group - 1) * NumSeeds) + 1;
        end_voxel = start_voxel + (NumSeeds - 1);
        % Store deviation data of all samples for the two voxels
        for Sample = 1:NumSamples
            for Vi = start_voxel:end_voxel
                VoxelDevData = VoxelDeviations(Sample, Vi);
                fwrite(SeedfileID,VoxelDevData,'float');
            end;
        end;

        % Store standard deviation data of the two voxels
        for Vi = start_voxel:end_voxel
            VoxelSDData = VoxelSDSum(Vi);
            fwrite(SeedfileID,VoxelSDData,'float');
        end;

        % Store voxel index
        for Vi = start_voxel:end_voxel
            fwrite(SeedfileID,Vi,'int');
        end;
    end;

    % Close the file
    fclose(SeedfileID);
end;
```

Figure 2.16: Preprocessing Step 7: Build seed data files (MATLAB Script)

Given the pairwise nature of the correlation coefficient (i.e., X correlates to Y and Y correlates to X), it becomes unnecessary to process the lower triangle matrix of the correlation matrix. Given

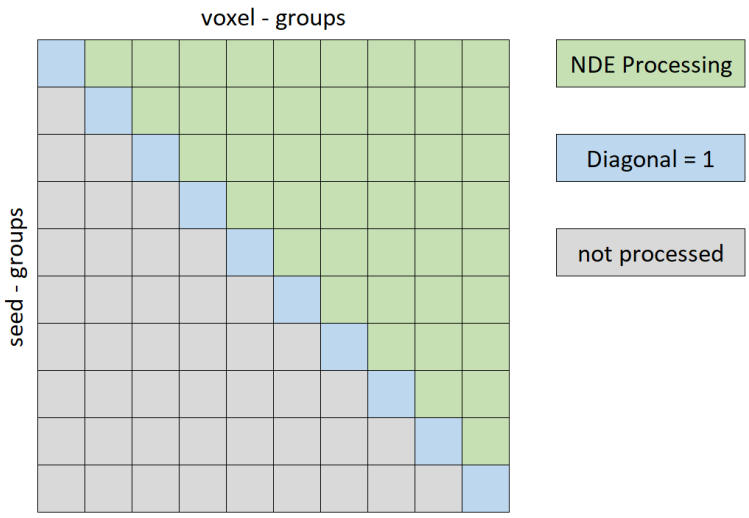


Figure 2.17: Upper Triangle Matrix - Correlation Matrix

this, and to create a condition where there is never a non-unique result, the workstation script during preprocessing computes the diagonal of the correlation matrix. Since this is a small number of voxels being processed, this process is typically fast. It should be noted, that nothing precluded the NDE from processing these regions, and could be enabled if the user so desired, but is unnecessary processing.

Processing tasks are generated that split the processing workload into groups of seeds to be correlated to a set of voxels. The object of the task generation script is to make the most efficient use of the processing cores across the given number of NDEs that the HPCME has. Tasks are processed into groups, and we divide the workload using the upper triangle matrix and starting with the upper left-hand corner working our way down. All processing is performed on the upper-triangle correlation matrix. Tasks are saved as a file for each NDE system and contain the voxel group, seed group, voxel offset, and r' to be processed. The CPU on the NDE is responsible for loading and monitoring the task processing. This task generation of the correlation matrix is illustrated in Figure 2.17 and is scripted as follows.

It should be noted that preprocessing can be performed without the need of the HPCME, thus enabling the HPCME to be processing correlation tasks all while new datasets are being preprocessed.

```
%--------------------------------------------------------------------------
% Build tasks
NumTasks = 0;
PrevNum = 0;
for voxelgroup = 1:NumGroups
    PrevNum = NumTasks;

    for seedgroup = 1:NumSeedsPerGroup
        voxeloffset = seedgroup * NumSeeds;
        voxelgroupStart = ((voxelgroup - 1) * NumVoxelsPerGroup) + 1;

        if (voxelgroupStart >= voxeloffset)
            offset = 0;
            NumTasks = NumTasks + 1;
            Tasks(NumTasks) = struct('VoxelGroup',voxelgroup-1,
                                     'SeedGroup',seedgroup-1,
                                     'Offset',offset);
        else
            offset = voxeloffset - voxelgroupStart + 1;
            if (offset < NumVoxelsPerGroup)
                NumTasks = NumTasks + 1;
                Tasks(NumTasks) = struct('VoxelGroup',voxelgroup-1,
                                         'SeedGroup',seedgroup-1,
                                         'Offset',offset);
            end;
        end;
    end;

    fprintf('Group %d Tasks\n', NumTasks - PrevNum);
end;

fprintf('Total Tasks %d\n', NumTasks);

% Get system level tasks
SysTasks = zeros(NumSystems, 1,'uint32');
for task=1:NumTasks
    System = mod((task - 1), 4) + 1;
    if (System <= NumSystems)
        SysTasks(System) = SysTasks(System) + 1;
        SystemTask(System, SysTasks(System)) = Tasks(task);
    end;
end;
```

Figure 2.18: Preprocessing Step 8: Build task scripts (MATLAB Script)

2.4 Memory Organization

Organizing the processing memory is one of the critical features for the NDE. For efficient streaming and sufficient processing speed, the data in memory must be in sequential order. Sequential memory access is one of the chief limitations of a CPU based system because instructions and data are located in the same memory, and depending upon the structure and the organization of the memory, effective and efficient bursting of SDRAM is not achieved. For instance, to get efficient bursting out of an SDRAM, it is best to burst a block of, at least, 64 sequential addresses at once, which is on the order of 512 bytes of data on a 32-bit bus and 1024 bytes on a 64-bit bus.

To have a consistent stream going through the processing core, the data is organized in such a way that samples are one after the other by the prescribed voxel's width, as illustrated in Figure 2.19. As described in the preprocessing section, the voxel data is stored sequentially, then followed by the standard deviation and the voxel indices.

The seed data is organized in a similar fashion, as previously discussed, and is illustrated in Figure 2.20. The seed index value, though stored in memory, is not utilized in the final computation of the r vector. However, it is maintained to enable full synchronization of the two data streams.

This memory organization enables the near continuous streaming of the data, sequentially, through the core. In fact, the only time that processing is halted is to stream the result vectors into SDRAM, and then to non-volatile storage, when the result BRAM becomes full. This transfer process is highly dependent upon the size of the data, r' threshold value, and the actual correlation connectivity.

Voxel Memory Map - SDRAM (DDR3)			
Address:	**Voxel #:**	**Sample #:**	
0	0	0	fMRI Sample Data
4	1	0	
8	0	1	
12	1	1	
16	0	2	
20	1	2	
24	0	3	
28	1	3	
...			
8424	0	1053	
8428	1	1053	
8432	0	1054	
8436	1	1054	
8440	0	StdDev	Standard Deviation
8444	1	StdDev	
8448	0	0	Voxel Index
8452	1	1	
next pair			
8456	2	0	fMRI Sample Data
8460	3	0	
8464	2	1	
8468	3	1	
8472	2	2	
8476	3	2	
8480	2	3	
8484	3	3	
...			
16880	2	1053	
16884	3	1053	
16888	2	1054	
16892	3	1054	
16896	2	StdDev	Standard Deviation
16900	3	StdDev	
16904	2	2	Voxel Index
16908	3	3	

Figure 2.19: Voxel memory organization

2.5 r vector and Results

Another crucial issue, beyond calculation efficiency, is in the storage of the resulting correlation matrix results. For a $91 \times 109 \times 91$ voxel data set (i.e., 902,629 voxels) and given the fact that P correlates to Q and that Q correlates to P, there are 4.07×10^{11} resulting correlation coefficients. If one stored each correlation coefficient as a 64-bit floating point r value, this would require 5.928

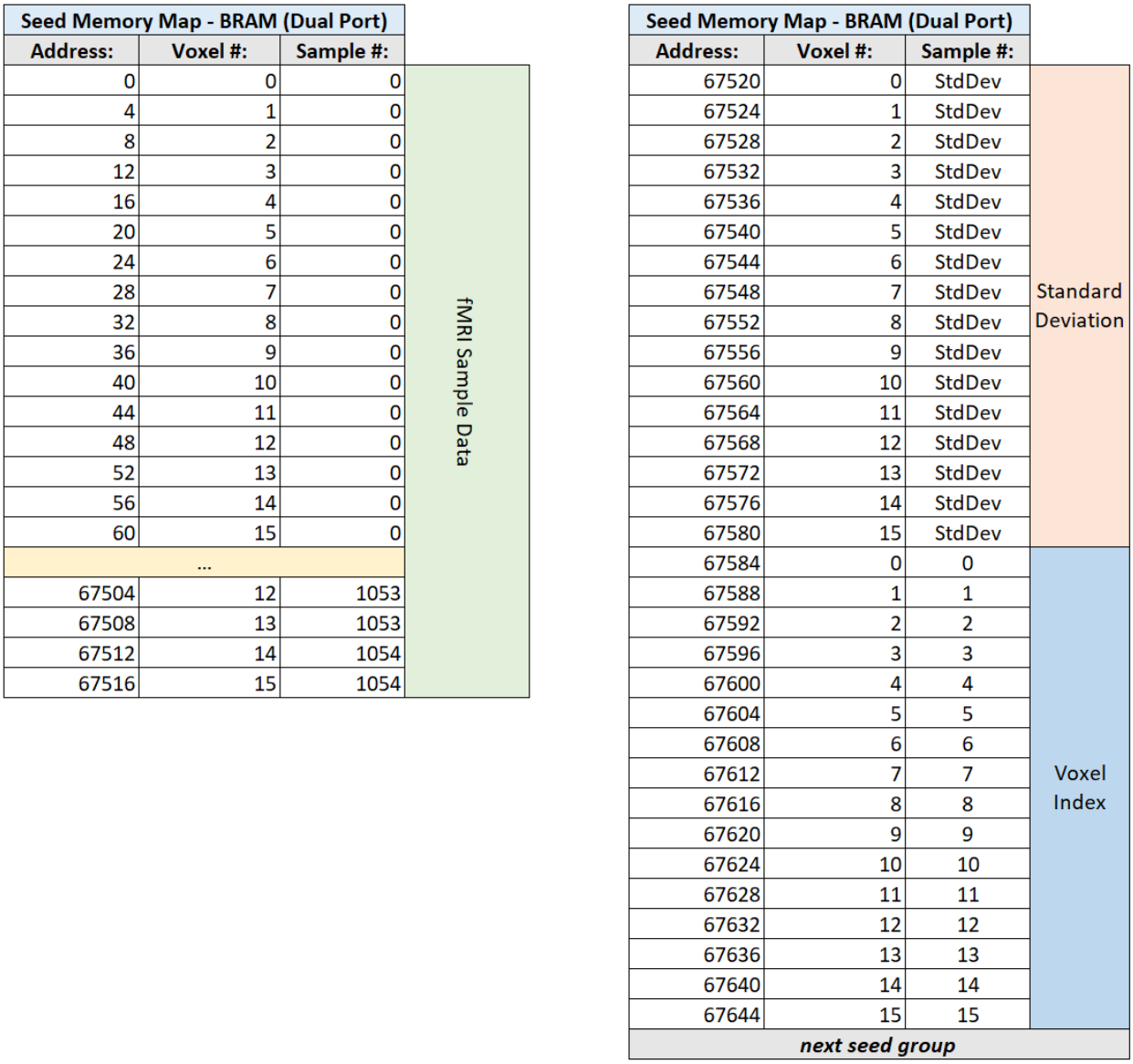


Figure 2.20: Seed memory organization

terabytes. This amount of storage per data set is not efficient or practical. This problem is resolved by the creation of a "correlation vector," which is a data structure between the seed and voxel. This vector generates for each connection that meets the minimum correlation threshold (i.e., r'). This correlation threshold is expected, based upon connectivity research, to create a connection density of about 2.8% to 3.0% [32]. This "correlation vector" significantly reduces the resulting data set to the order of megabytes, and not terabytes, depending upon the value of this threshold. The format of the "correlation vector," shown in Figure 2.21, has a length of 8 bytes (64-bits), and in post-processing reduces to a simple connection state depending on the final desired output.

Correlation Vector		
32-bits	27-bits	5-bits
Floating Point	integer	integer
Correlation Coefficient	**Voxel #**	**Seed Index**

Figure 2.21: Correlation r Vector (Result Vector)

2.6 Postprocessing Steps

Postprocessing of the results for HPCME consists merely of importing the r vectors that are stored on each NDE. Using this data one may generate the final merged r vectors and then generate connectivity and connectedness "heat" maps.

```
// Process r-Vector Files
if (File.Exists(fileName) == true)
{
    // Open a file stream to the file and binary reader
    FileStream fs = File.Open(fileName, FileMode.Open);
    using (BinaryReader br = new BinaryReader(fs))
    {
        // Process until the EOF
        while (fs.Position < fs.Length)
        {
            // Read results header
            int iVoxelGroup = br.ReadInt32();
            int iSeedGroup = br.ReadInt32();
            int iOffset = br.ReadInt32();
            int iNumBytes = br.ReadInt32();
            int iNumResults = iNumBytes / 8;

            // Read data, make sure mod of 8 bytes (i.e. result vector packet is 64 bits)
            if ((iNumBytes % 8) == 0)
            {
                // Step through all the results
                for (int iRes = 0; iRes < iNumResults; iRes++)
                {
                    // Read "r" value and index
                    // Read floating-point value (Correlation Coefficient)
                    float rValue = br.ReadSingle();

                    // Read the 32-bit Index Value
                    int rIndex = br.ReadInt32();

                    // Get the Voxel Index for this Coefficient - (upper 27 bits)
                    int iVoxelIndex = rIndex >> 5;

                    // Get the Seed Index, based on seed group in header,
                    // number of seeds, and stored index
                    int iSeedIndex = (iSeedGroup * iNumSeeds) + ((rIndex >> 1) & 0xF);

                    // Increment connectedness map by the indices
                    voxelmap[iVoxelIndex]++;
                    voxelmap[iSeedIndex]++;

                    // Do other analysis as desired...
                }
            }
        }
    }
}
```

Figure 2.22: r-Vector Results Processing (C#)

To create the merged actual r vector, the postprocessing script shown in Figure 2.22, must first compute stored voxel and seed index into the actual voxel and seed index. Converting to these indices is a simple process as the seed index is based upon the sub-seed index and seed group index, which is part of the processing results. The r index is made up of a voxel index and the seed index determines which of the group of seeds the r value corresponds.

The voxel index is the upper 27 bits of the r index value and corresponds to which the actual voxel, where the r vector is assigned.

Postprocessing this r vector data is processed to generate a voxel map which can either be displayed as connections between the seed and the voxel or as a heat map. In the case of generating a connectedness heat map, one just merely counts how many connections that a particular voxel was connected, then based upon this, determines a normalized heatmap. The heat map is based on using the Node Degree, which as defined previous, is based upon the total number of connections for a particular voxel, or node, P is simply:

$$k_P = \sum_{i=1}^{M} e_{P,i} \tag{2.4}$$

With completeness as a metric that represents the value of connections for a particular voxel versus the maximum number of total observed connections of the whole brain.

$$Completeness = \frac{k_P}{MaxConnections} \tag{2.5}$$

The postprocessing of the r vector data is processed rapidly and does not require a tremendous amount of processing resources. This fact is chiefly because the r vectors only correspond to a small percentage of the original correlation matrix.

```
%--------------------------------------------------------------------------
% Import file normalized result file
FileName = sprintf('%d_map.bin', SubjectNumber);
ResultfileID = fopen(FileName,'r');
FileResults = fread(ResultfileID,Inf,'double');
fclose(ResultfileID);

%--------------------------------------------------------------------------
% Get subset (i.e. orgnial data size)
Results = FileResults(1:NumVoxels)';


%--------------------------------------------------------------------------
% Reshape the data back to 3D
Results = reshape(Results,size(NIfTIDataset,1), size(NIfTIDataset,2),size(NIfTIDataset,3));

%--------------------------------------------------------------------------
% Display heatmap 3D (3rd party toolbox
viewer3d(Results);


%--------------------------------------------------------------------------
% Save images to file
maxval = max(max(max(Results)));
for z=1:1:size(NIfTIDataset,3)
    colormap(hot);
    FileName = sprintf('%d_connectedness_zslice_%d.png', SubjectNumber, z);
    imwrite(Results(:,:,z).* (255.0 / maxval), FileName);
end;


%--------------------------------------------------------------------------
% Save results to mat file (for additional analysis
FileName = sprintf('results_%d.mat', SubjectNumber);
save(FileName,'Results');
```

Figure 2.23: Postprocessing connectedness data (MATLAB Script)

3.1 Architecture and Implementation for PoC System

A Xilinx UltraScale KCU105 evaluation board [33] was initially used to test concepts and serve as a platform to process 64 seeds against four voxels in one larger FPGA. In theory, the specification of the evaluation board along with the capability of the XCKU040 Kintex FPGA [34] would serve this function well. However, upon implementation of the VHDL code in the Vivado development software, it became apparent that the extensive bus networks (up to 2,048 bits) were proving too much for the initial FPGA design to meet the timing objectives of a 250 MHz design. Various attempts were made to meet timing, and though there appear to be some

Figure 3.1: Xilinx KCU105 Evaluation Board

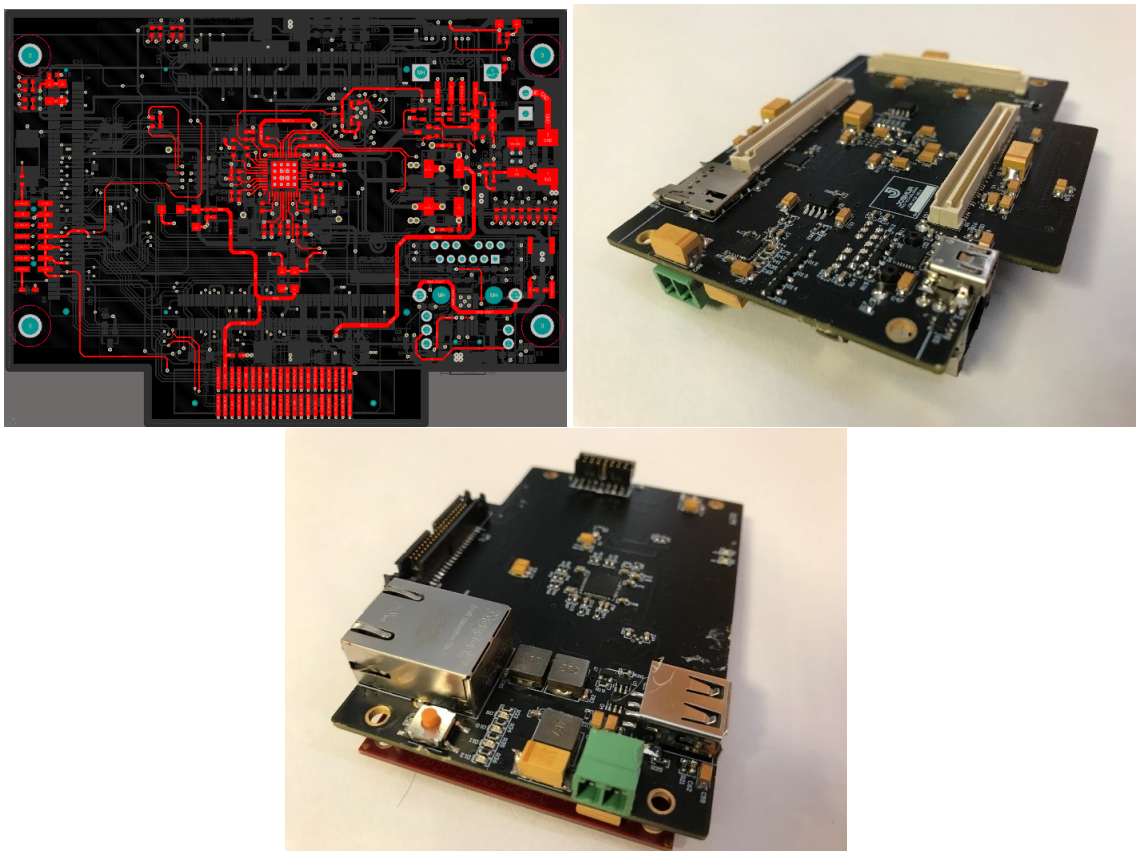


Figure 3.2: Top design layer of interface PCB in Altium Designer and Assembled Interface PCB

promising methods to get timing closure, it was taking too much development time and effort to be practical.

An alternative was found in the Avnet PicoZed Zynq Kintex 7Z030 System-On-Module (SoM) [35]. This SoC module was selected to become the new FPGA processing core for the Node Degree Engine and a significant component in the HPCME. Though the Zynq 7Z030 had functionally fewer gates, it did have the built-in bus management system that the SoC (System-On-Chip) provides. These additional resources helped meet the need for memory overhead and timing closure. An initial test project was created, and the synthesis and implementation passed, with timing closed, for a 200 MHz design.

The PicoZed Zynq 7030 SoM needed supporting hardware to function such as a power supply rails, a communications system, and an SD Card interface. This supporting hardware was designed to host the PicoZed. The electrical schematic and PCB (Printed Circuit Board) was developed in

Altium Designer [36]. This interface PCB provided the necessary power for the FPGA, Ethernet, USB, and UART communications. It also included a microSD interface to load and save files to a FLASH card [37]. Various LEDs and LVDS [38] ports were also added to enable hardware debugging as necessary. Various images of the design and completed and assembled PCB is in Figure 3.2.

A fully assembled system was verified by running it through a set of power-up and communication tests. This testing showed that the power supplies operated at the proper voltages and power levels. Interface testing ensured that the UART, Ethernet, USB and microSD card was fully operational as a subsystem component.

A test application was written in C and was designed to run in one of the two Zynq ARM processors [39]. This application was used to verify the base performance of the VVCA core and necessary support hardware. Verification was done by writing to memory multiple test voxels and seeds and examining the results and overall data throughput, which is presented later in this paper.

The first NDE system, shown in Figure 3.3, was loaded with the NDE hardware BIT file and tested with the test vector application previously mentioned. The test proved to be successful and demonstrated that the simulated and projected results did operate at the expected 200 MHz processing rate, minus some skipped cycles for operating overhead, such as SDRAM memory refresh.

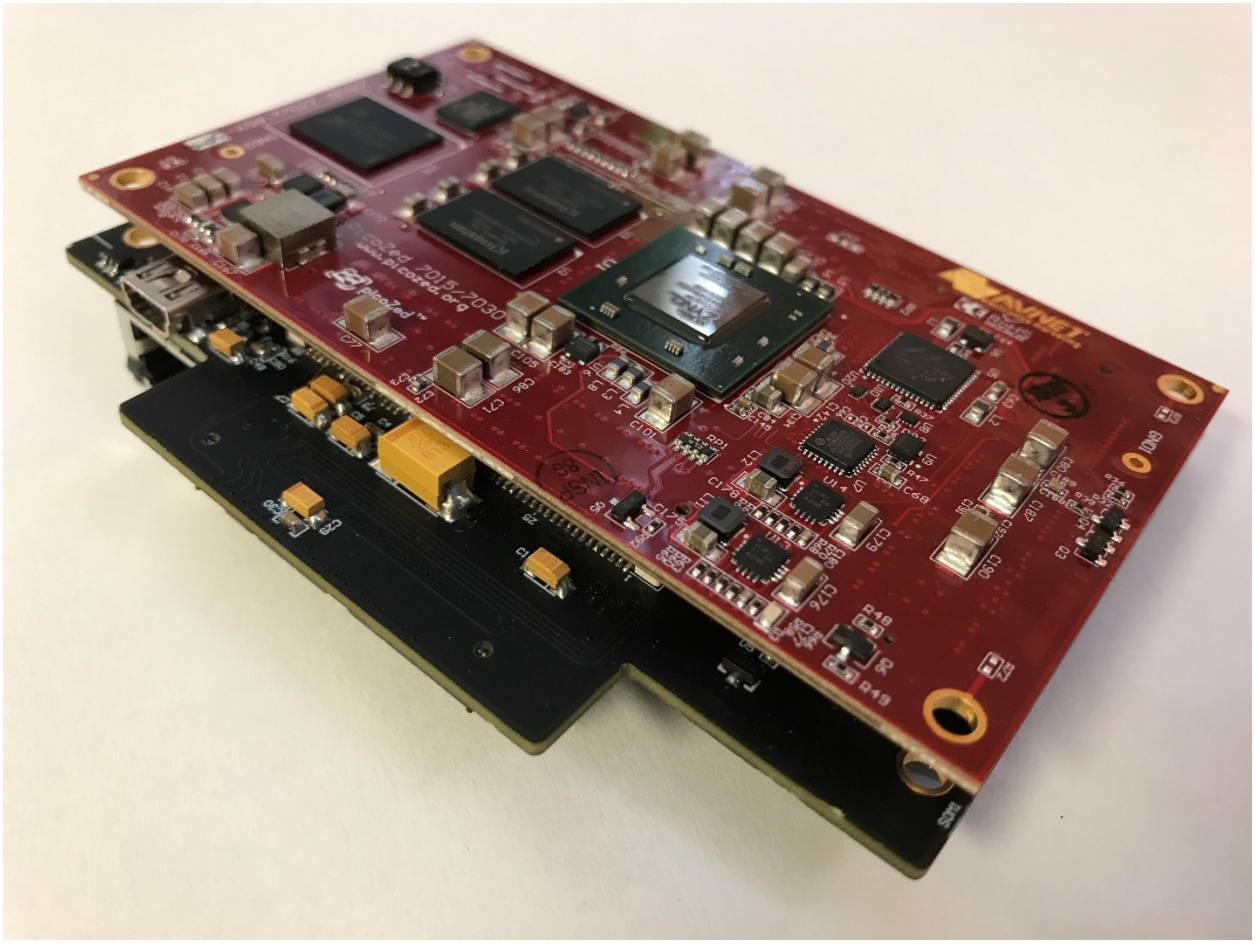


Figure 3.3: Prototype of the NDE

3.2 Logic and Firmware Implementation

The NDE based on the Zynq 7Z030 can only do 64 simultaneous correlations versus the originally planned 256. However, this modular based system using four Zynq 7Z030s can replace the performance of one XCKU040 at the same cost, if not less. The system allows for scaling, and if one uses eight NDEs, one will be able to process 512 simultaneous correlations versus the originally planned 256.

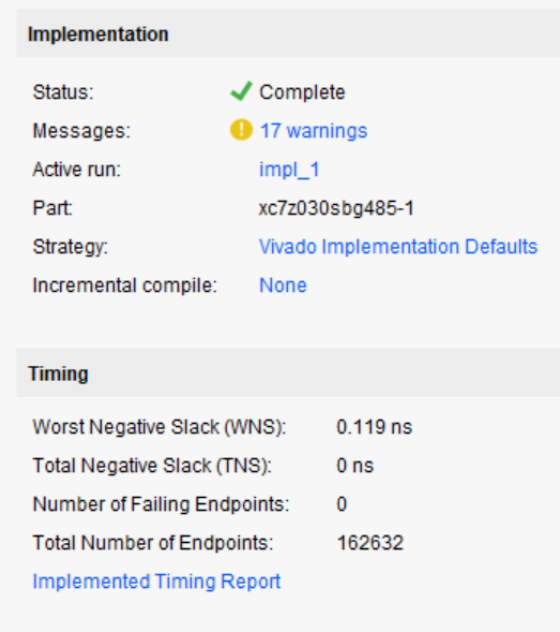


Figure 3.4: Xilinx Zynq 7Z030 implementation results of 64 Core NDE System

The initial coding of the CCA core was written in VHDL and integrated with the various supporting DMAs and memory subsystems necessary for the process to function at the expected data rate. The design block diagram is in 3.6. This block diagram features the two ARM processors, a 64-bit high-performance memory interface with a DMA to stream two voxels at 200 MHz. There is also a BRAM that is 1,024 bits wide enabling the storage of 32 seeds. A DMA is used to stream, at 1,024-bit wide, the seeds to the CCA core, also at 200 MHz. The CCA core generates results

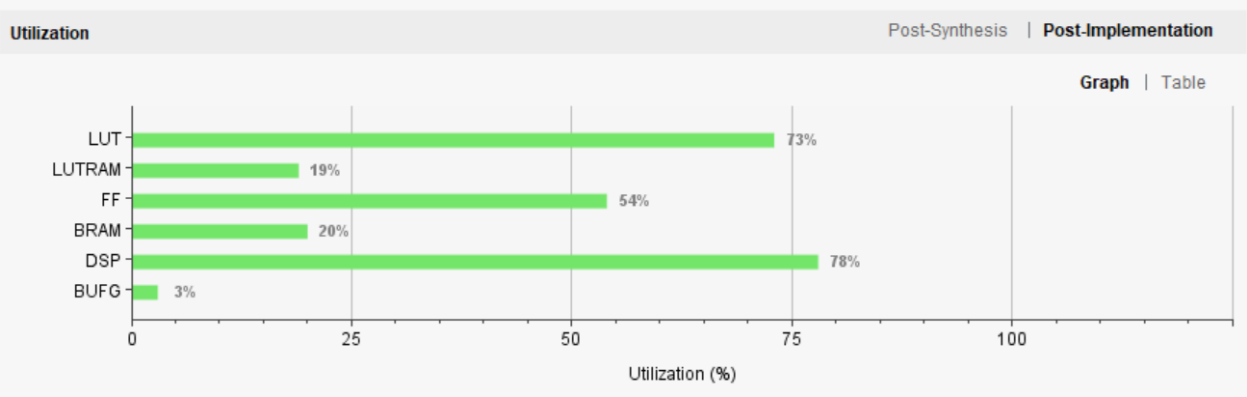


Figure 3.5: Xilinx Zynq 7Z030 resource utilization of 64 Core NDE System

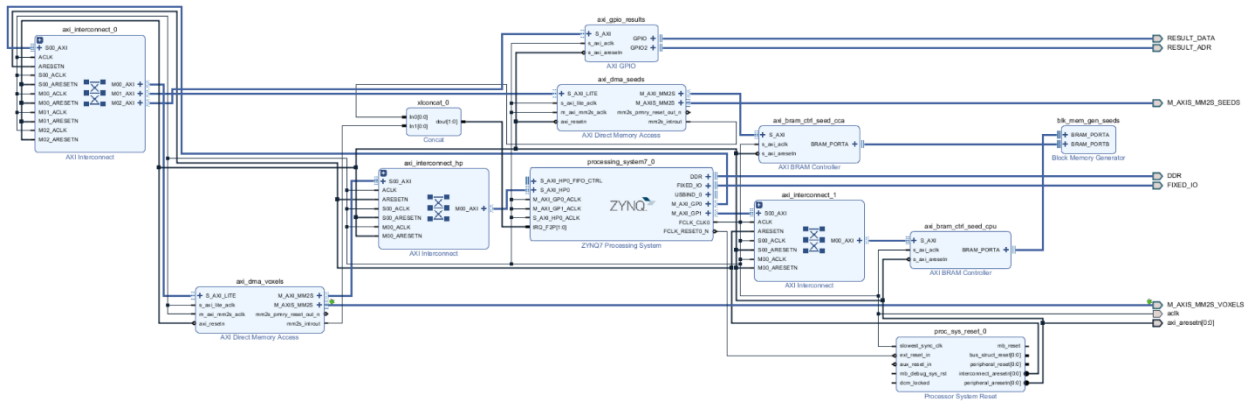


Figure 3.6: Initial PoC NDE - logic block diagram

that are then post-processed for the correlation coefficient.

The project with 64 simultaneous correlations was implemented successfully, with some timing modifications made to various bus interconnects so that timing closed. The resulting resource utilization in Figure 3.5, demonstrated that there was enough room for additional CCA processing or other operations. The implemented timing summary, shown in Figure 3.4, illustrates the timing closure and that the gate layout and implementation were successful.

The firmware and VHDL RTL coding for this Proof-of-Concept NDE have not been included in this section. However, the final coding is included in Section 4.2.

3.3 Results of Initial Tests

A representative set of results from testing the NDE with synthetic datasets are shown in this section. These tests were performed with the Zynq 7Z030 SoC FPGA and with the data being streamed at a conservative 200 MHz clock rate, with just one NDE core. A set of sixteen tests were conducted with similarly sized synthetic HCP data sets. These datasets contained 1,059,968 voxels, doing 25% of the possible assigned seeds, and with 1,024 samples each leading to 5.617×10^{11} possible node connections. The correlation coefficient was chosen to provide an 8% connectivity, versus the normally expected 2% to 3%. The more extensive connectivity was selected to understand the storage requirements and storage time.

Figure 3.8 shows the total processing and storage time for the different number of seed groups. Each group contains 32 seeds and the entire time increases monotonically until the final group is processed. Results showed a data rate of approximately 1.445 TFLOPS. Of note is that this processing rate is sustained throughout processing and not just in processing short bursts, as the bottleneck, with most processing systems, is not necessarily the ALU (Arithmetic Logic Unit) but the memory bandwidth to and from the ALU core. This architecture system resolves this issue by having a nearly dedicated memory architecture to maximize the processing of these large 4D data sets.

In Figure 3.9 the individual time of processing and the storage of the results vectors to the FLASH media is shown. This chart shows that as the processing group increments, the process and

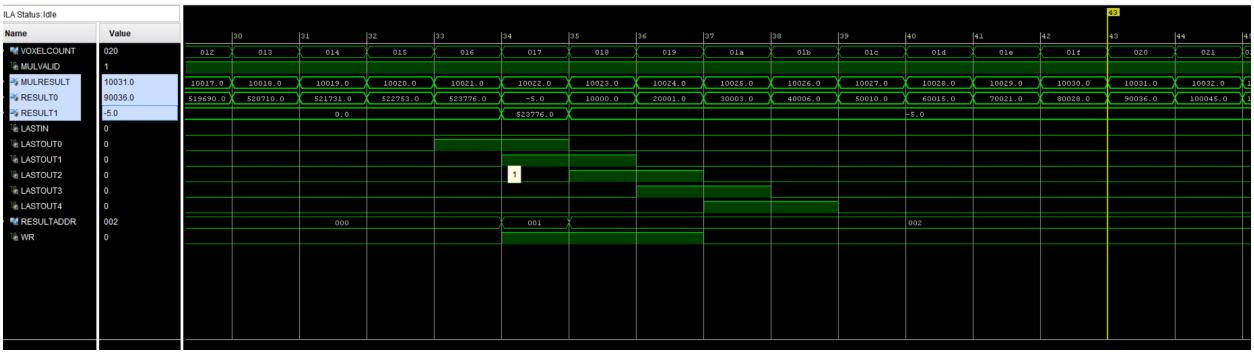


Figure 3.7: Logic analyzer - development testing of initial core

storage time decreases. This decrease is because of the reduced number of nodes being processed.

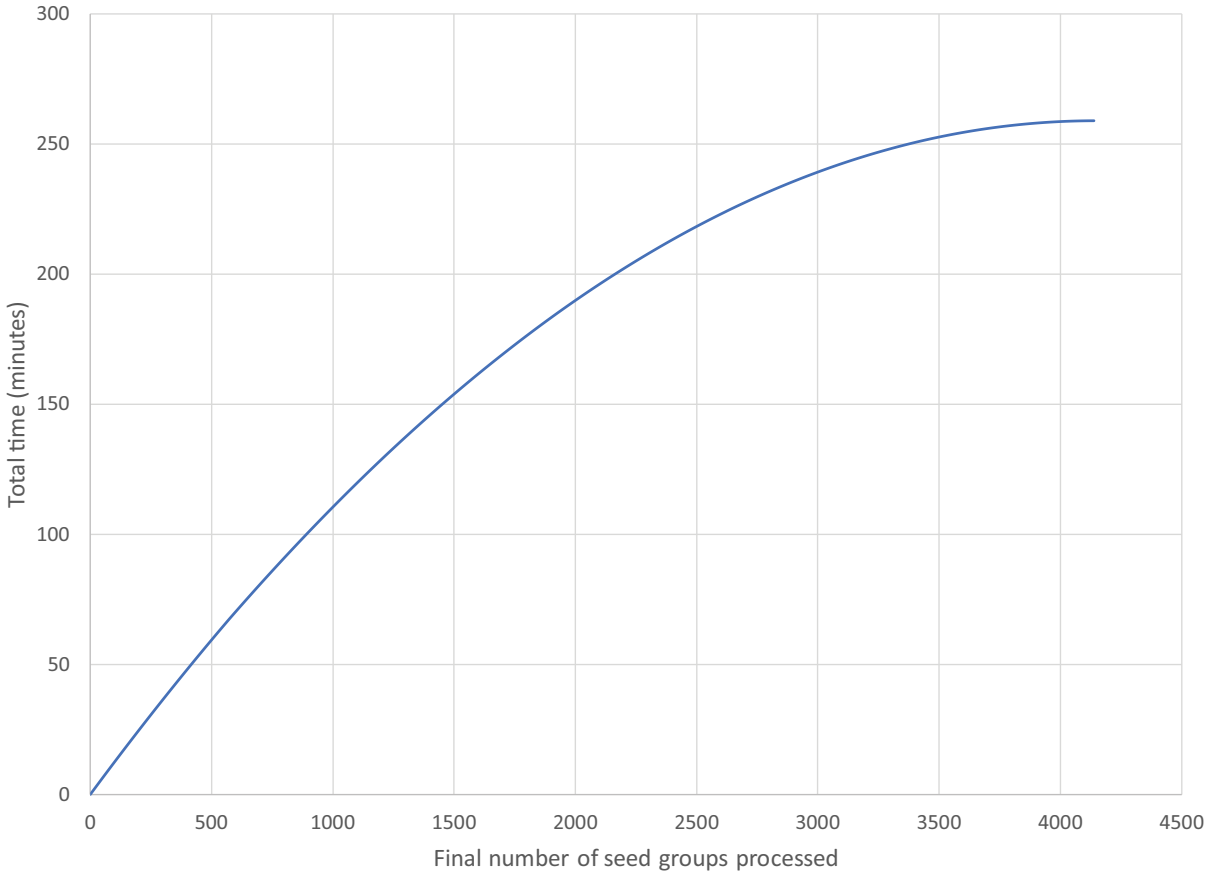


Figure 3.8: Total processing and storage with cumulative seed groups for a data set of $128 \times 91 \times 91 \times 1024$

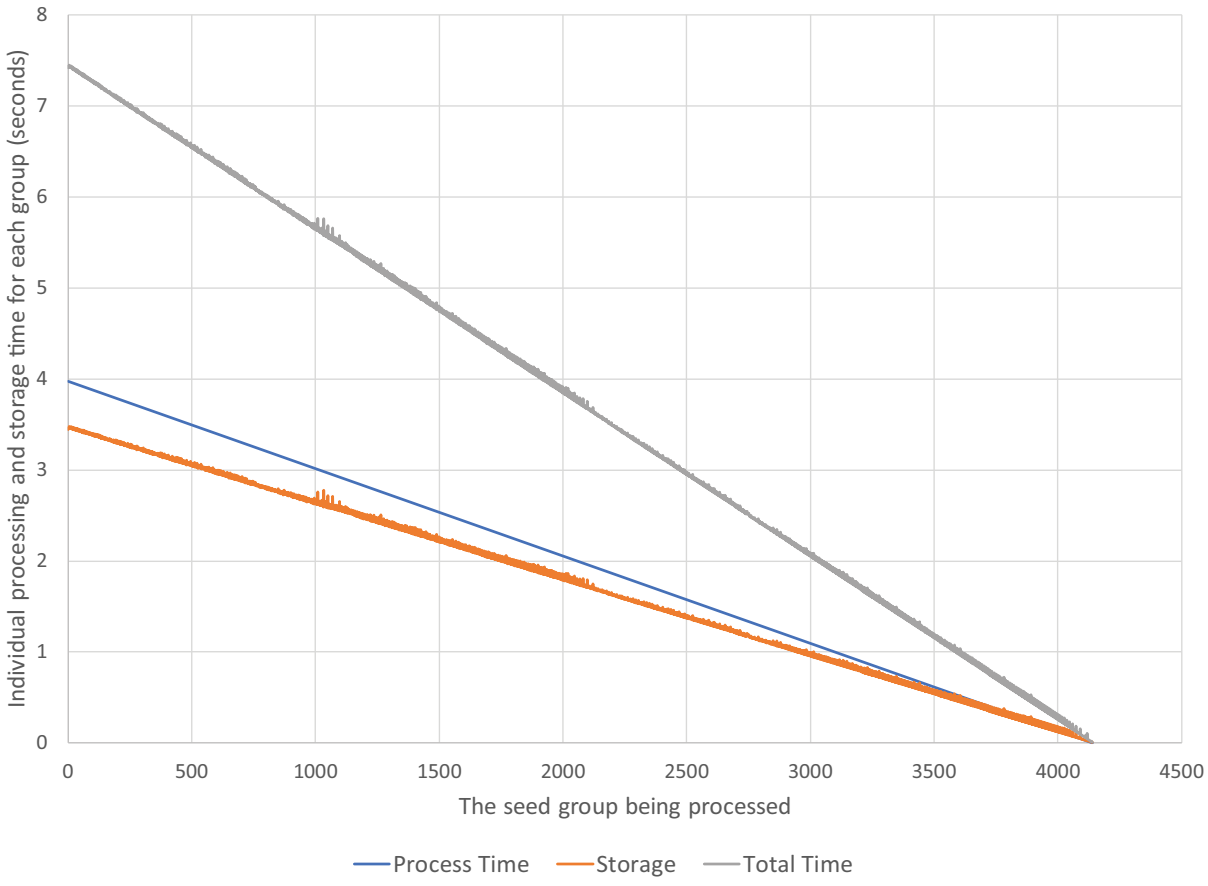


Figure 3.9: Individual processing and storage time for each seed group

3.4 Initial HCP Test and Discussion

The HPCME system was initially tested using a HCP dataset provided from the Human Connectome Project. This dataset has a resolution of 91 voxels in the coronal plane, 109 voxels in the sagittal plane, and 91 voxels in the transverse plane. This resolution provides a dataset of 902,629 voxels with a sample size of 284 samples.

This dataset was preprocessed and organized as described in the preceding preprocessing section and the resulting files copied to a FLASH SD card for utilization by the HPCME system. Task files were also generated for each NDE in the system, and with the optimization that this NDE have ¼ of the task load.

After the files were loaded into the HPCME, an application running on a PC workstation laptop was activated and the power applied to the HPCME. Communications were established with the HPCME, and the task processing system was initiated.

It should be noted, that it is possible with the HPCME that the logging and monitoring of the results are not required during task processing of the data, as the HPCME does not require operator intervention in processing. However, for this specific application, all datasets being processed were monitored and the corresponding processing times recorded along with the final result vector files.

This one HCP dataset was processed five times and the results of the process show an average processing time of 1.45 hours, with a standard deviation of 48 seconds. The connectedness map of this HCP dataset is shown in Figure 3.10.

These initial results of this Proof-of-Concept NDE showed that this method could potentially produce high-resolution brain connectivity maps rapidly. The new method accelerates the correlation processing by using an architecture that includes clustered FPGAs and an efficient memory pipeline. The preliminary results showed that HPCME with four FPGAs, using this 64 core approach, could improve the VVCA processing speed by a factor of 82 or more over that of a traditional PC workstation with a multicore CPU. These comparisons will be explored further in the following sections.

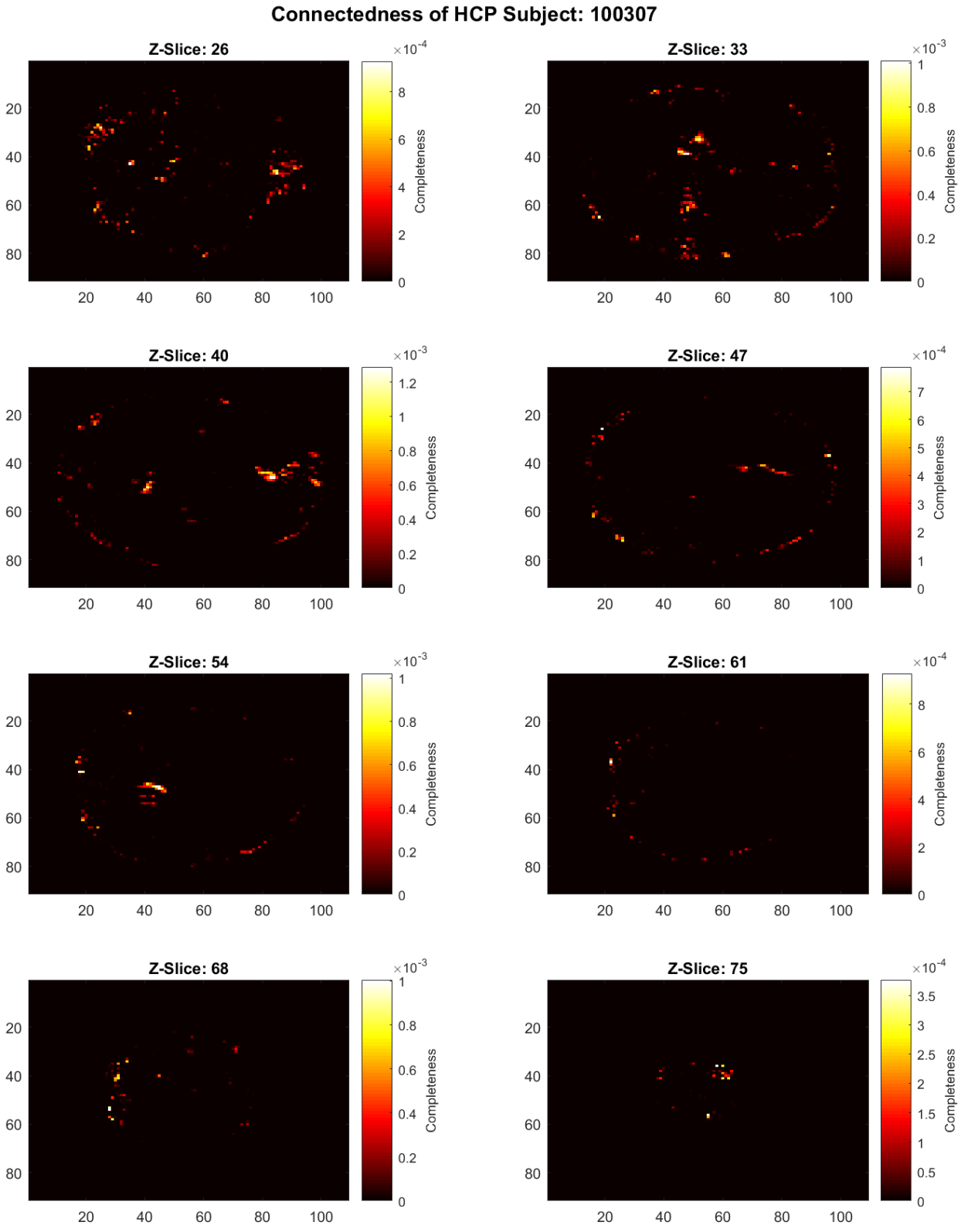


Figure 3.10: Initial NDE PoC connectedness map of HCP Subject #100307

4.1 Architecture and Implementation System

After the initial development and testing of the single Proof-of-Concept Node Degree Engine (NDE), the HPCME system implementation was developed. The HPCME for this research was a group of four NDEs combined into one complete processing system. Each NDE, being about the size of a smartphone, was arranged in a desktop enclosure (from PacTec) and would run indepen-

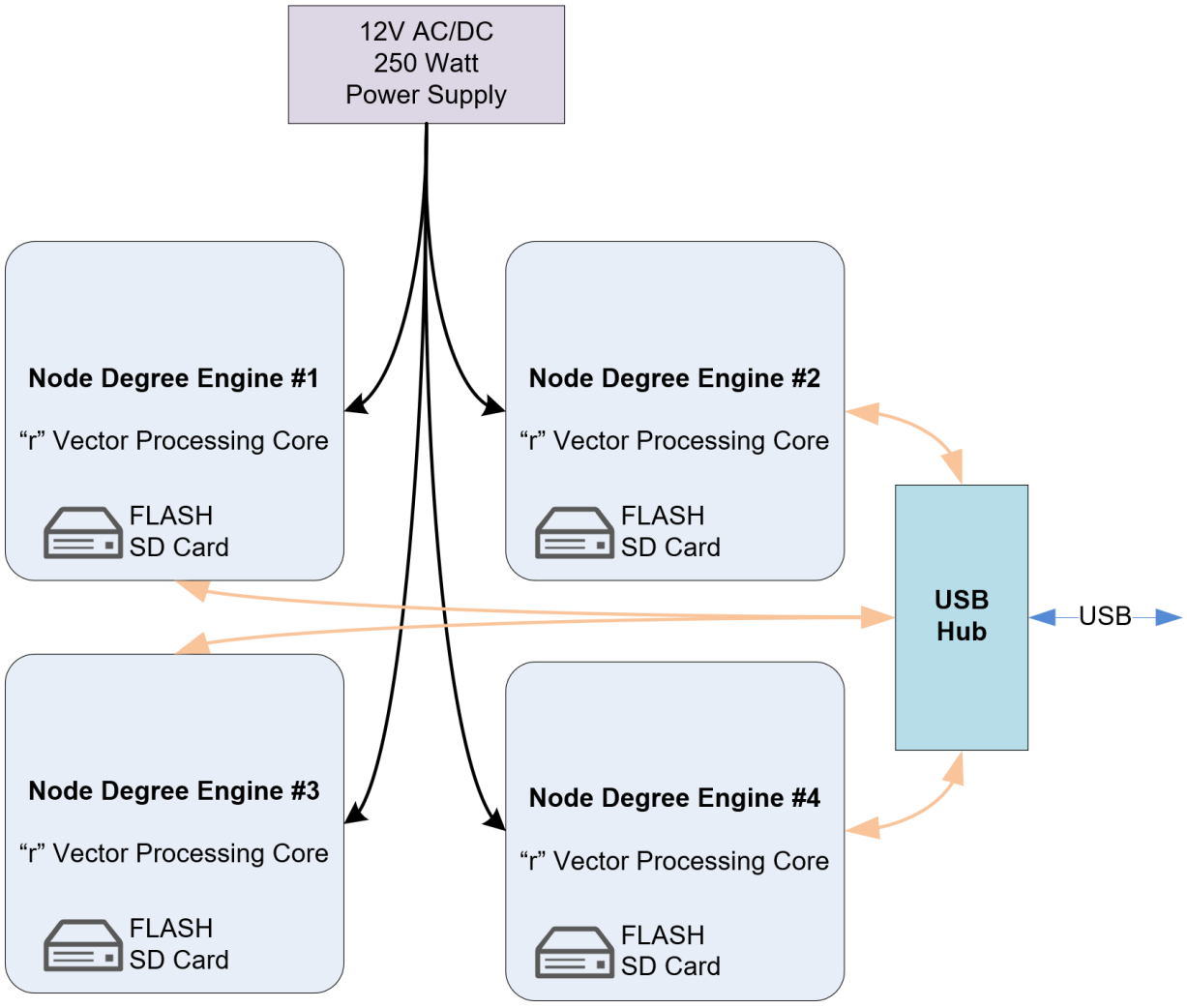


Figure 4.1: The HPCME System - Block diagram layout

Figure 4.2: The HPCME System - Implementation

dently from each other, enabling near complete parallel operations.

The HPCME utilized a shared power supply and a small USB hub that concentrated the USB communication for the NDEs. A thermal management system (i.e., vents and a high-speed fan) is used to expel the heat generated by the FPGAs during processing. Each NDE has an external SD card for use as storage for the HCP datasets, tasks, and the final result vectors. The HPCME configuration is shown schematically in Figure 4.1 and a photograph of the actual implementation in Figure 4.2.

To begin processing HCP data, a MATLAB script is utilized to preprocess all HCP datasets into tasks, seed, and voxel data files which are then transferred to the designated SD cards for processing. This script is shown in detail in Section 2.3.

A windows based PC workstation application was developed to provide a simplistic interface for the HPCME. This application, written in Microsoft Visual Studio C#, enabled user status updates, logging, and control over the operation of the HPCME. The application will provide the user feedback to which tasks are currently being processed on each NDE, including the current

seed and voxel group, elapsed time, and the estimated time remaining. A user will also be able to initiate or stop task processing. This application interface is shown in Figure 4.3.

Once the HPCME system has completed the task processing, the results are transferred to a PC workstation for post-processing. The post-processing is done with the aid of the very same Microsoft Visual Studio C# application which rapidly reads the result files and computes the connectedness, as described in Section 1.3. Then, a MATLAB script is utilized to perform chart and visualization of the resulting r-vector datasets. This MATLAB script is shown in Figure 4.4.

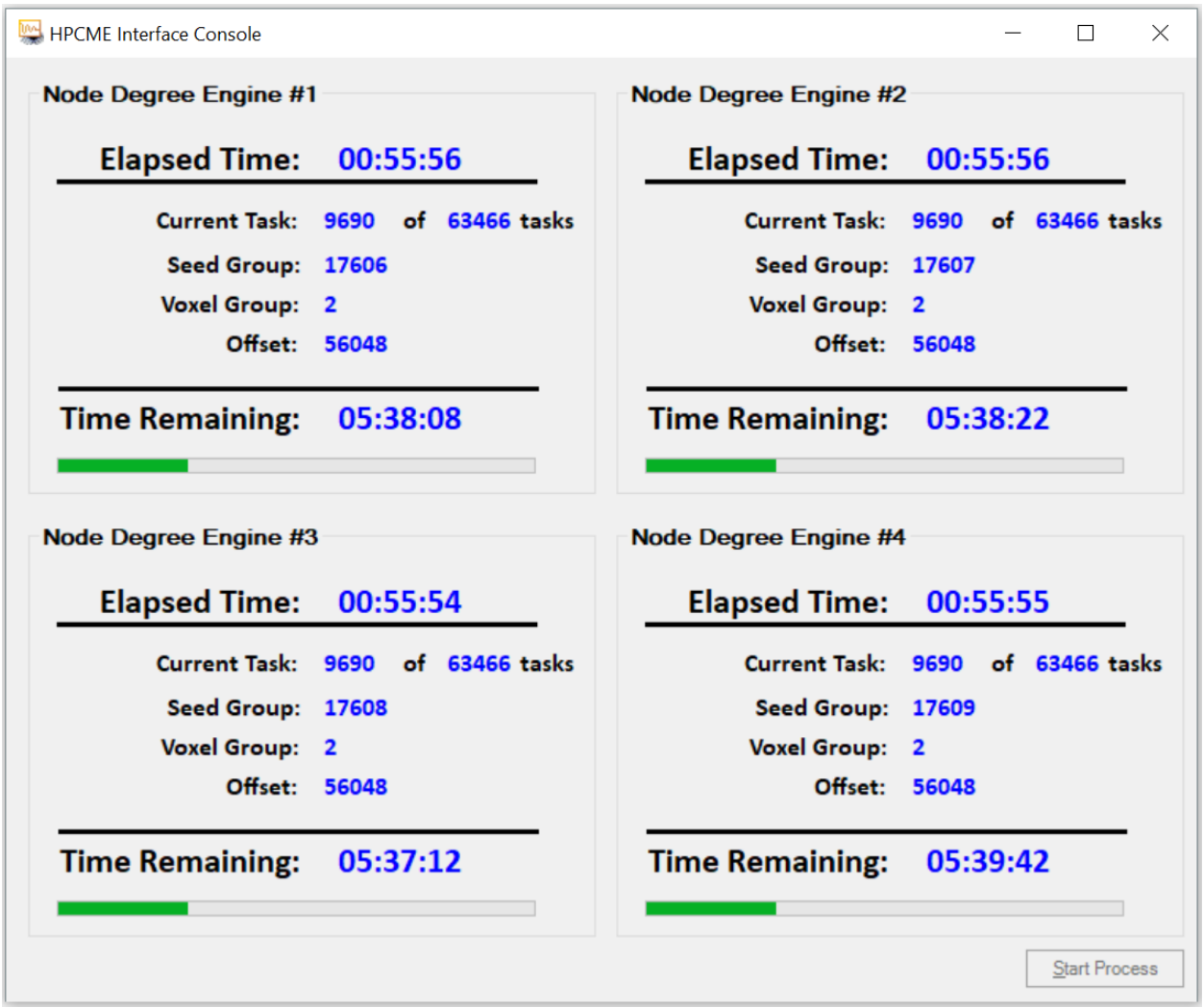


Figure 4.3: HPCME interface and control software

```
%--------------------------------------------------------------------------
% Import file normalized result file
FileName = sprintf('%d_map.bin', SubjectNumber);
ResultfileID = fopen(FileName,'r');
FileResults = fread(ResultfileID,Inf,'double');
fclose(ResultfileID);

%--------------------------------------------------------------------------
% Get subset (i.e. orgnial data size)
Results = FileResults(1:NumVoxels)';


%--------------------------------------------------------------------------
% Reshape the data back to 3D
Results = reshape(Results,size(NIfTIDataset,1), size(NIfTIDataset,2),size(NIfTIDataset,3));

%--------------------------------------------------------------------------
% Display heatmap 3D (3rd party toolbox
viewer3d(Results);


%--------------------------------------------------------------------------
% Save images to file
maxval = max(max(max(Results)));
for z=1:1:size(NIfTIDataset,3)
    colormap(hot);
    FileName = sprintf('%d_connectedness_zslice_%d.png', SubjectNumber, z);
    imwrite(Results(:,:,z).* (255.0 / maxval), FileName);
end;


%--------------------------------------------------------------------------
% Save results to mat file (for additional analysis
FileName = sprintf('results_%d.mat', SubjectNumber);
save(FileName,'Results');
```

Figure 4.4: HPCME NDE postprocessing of connectedness data (MATLAB Script)

4.2 Logic and Firmware Implementation

For the full HPCME implementation of the NDE, it was decided to use a core with 32 simultaneous correlations versus the previous tested 64. This change was done to enable the addition of diagnostic hardware, including a logic analyzer. This implementation, with four NDEs in the HPCME system, enables a total of 128 simultaneous correlation cores, versus the previous 256 cores.

The project with 32 simultaneous correlations was implemented successfully, with some timing modifications made to the various bus interconnects so that timing closed. The resulting resource utilization in Figure 4.6, demonstrated that there was enough room for additional VVCA processing or other operations. The implemented timing summary, shown in Figure 4.5, illustrates the timing closure and that the gate layout and implementation were successful.

The coding of the VVCA core was written in VHDL and integrated with the various supporting DMAs and memory subsystems necessary for the process to function at the expected data rate (200 MHz). The NDE design block diagram is in Figure 4.7. This block diagram features the two ARM

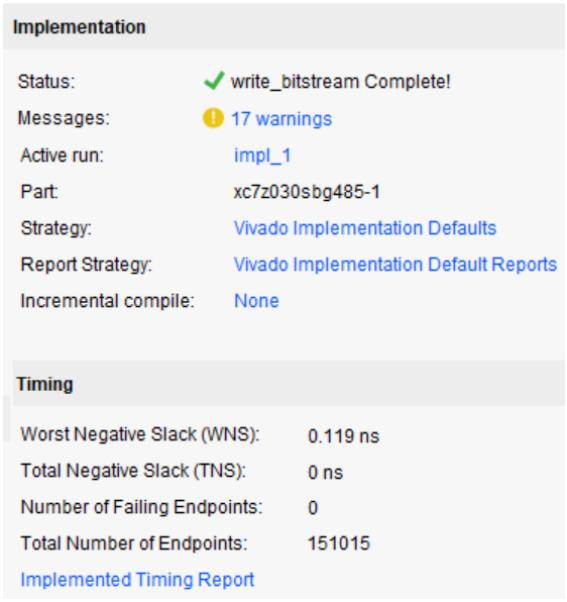


Figure 4.5: Xilinx Zynq 7Z030 Implementation Results of 32 Core NDE System

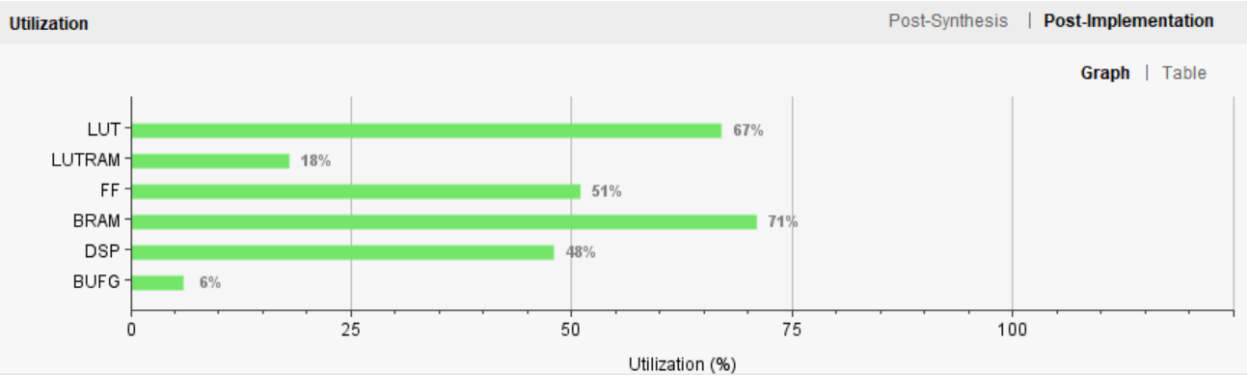


Figure 4.6: Xilinx Zynq 7Z030 Resource Utilization of 32 Core NDE System

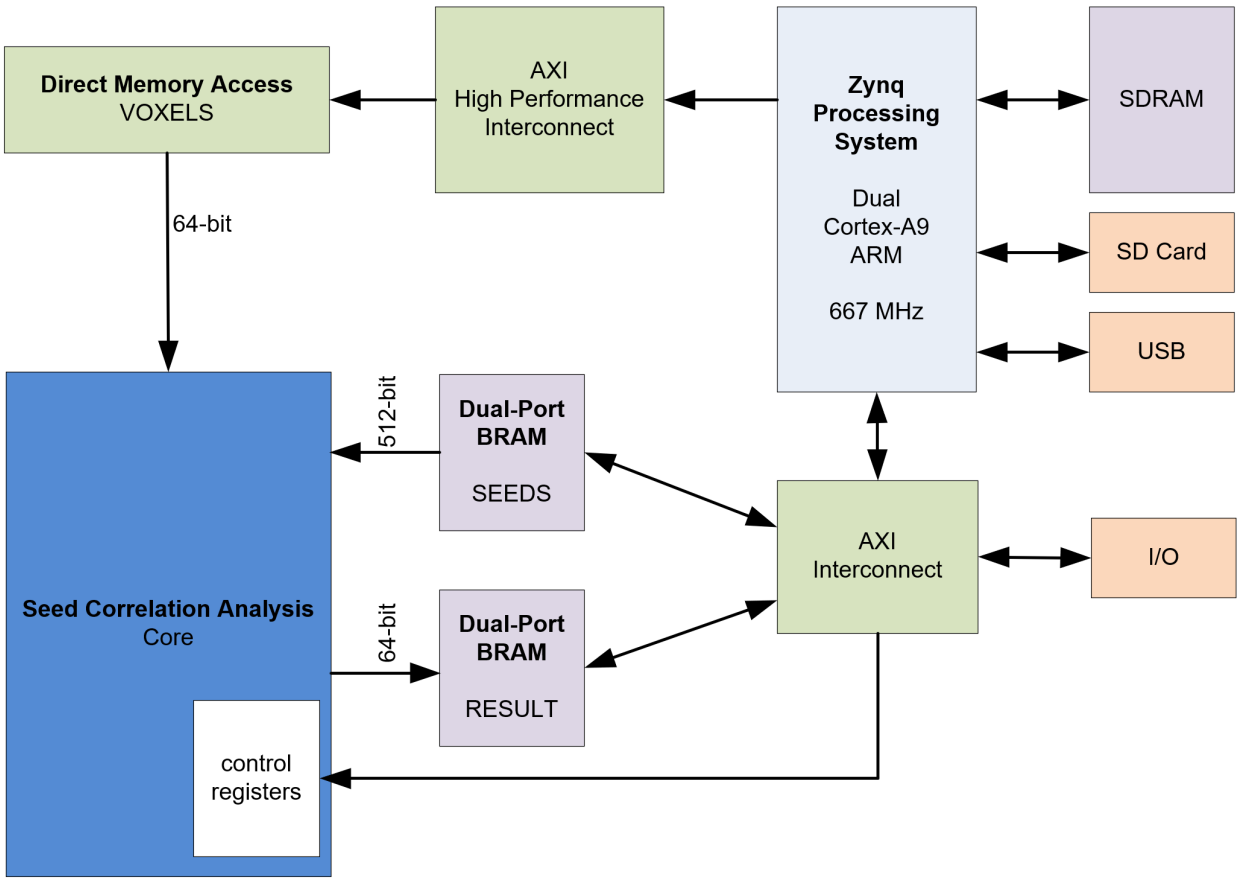


Figure 4.7: HPCME NDE - logic block diagram

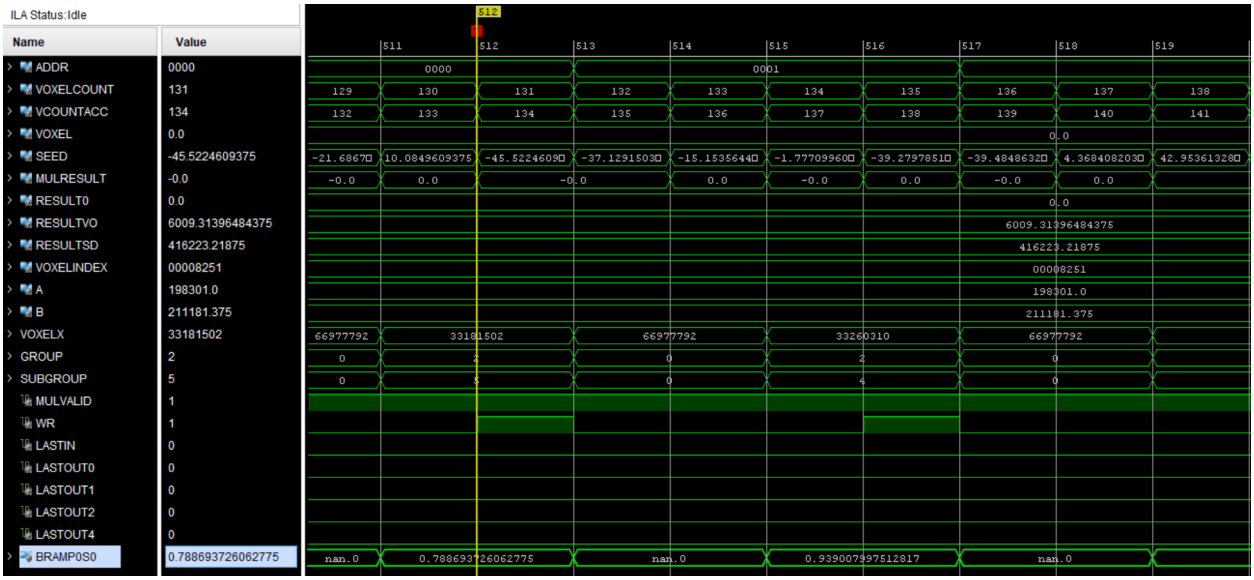


Figure 4.8: HPCME NDE - Timing diagram - Results write to BRAM (ILA Logic Analyze)

Cortex A9 processors, a 64-bit high-performance memory interface with a DMA to stream two voxels at 200 MHz. There is also a dual-port BRAM that is 512 bits wide enabling the storage of 16 seeds. A DMA is used to stream, at 512-bits wide, the seeds to the VVCA core, also at 200 MHz. The VVCA core generates results that are then post-processed for the correlation coefficient as discussed in previous sections. The timing diagrams and SoC schematic are shown in Figures 4.8, and 4.9 respectively.

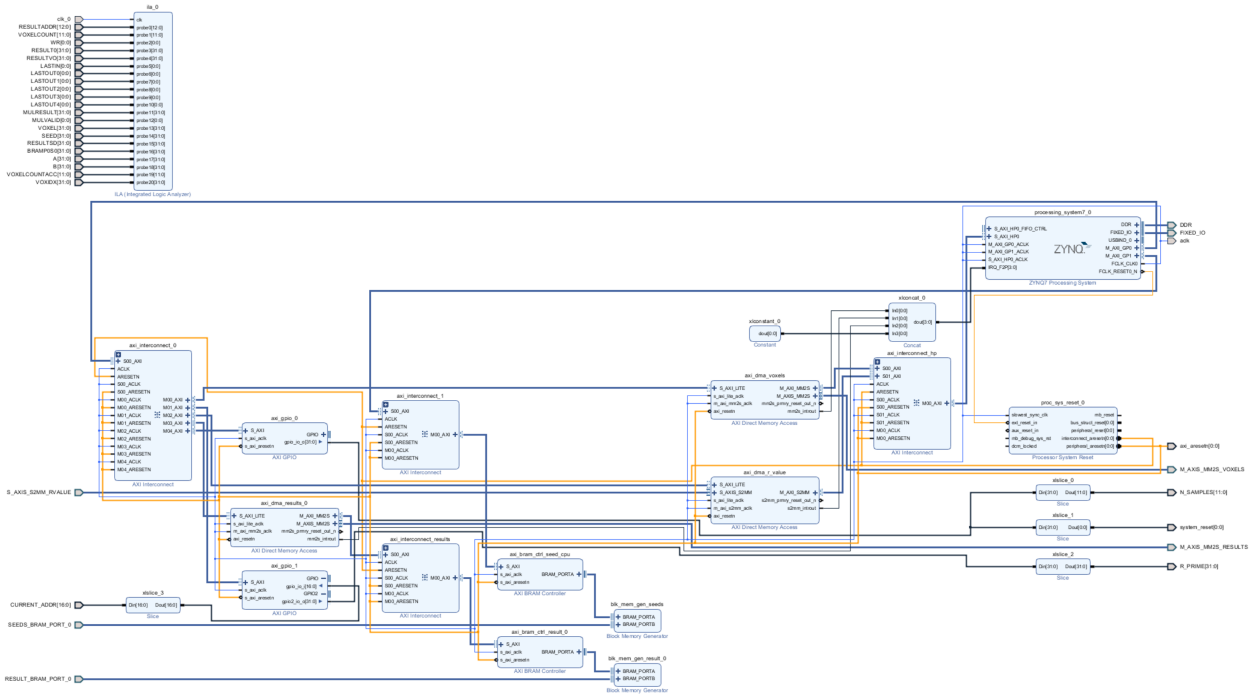


Figure 4.9: HPCME NDE - Logic schematic (Xilinx Vivado Block Diagram)

The critical sections of the VHDL implementation of the Seed Correlation Analysis is listed as follows.

```
entity cca_core is
    Generic
    (
        DATA_WIDTH_BYTES                : natural := 4;
        DATA_WIDTH_BITS                 : natural := 32;
        NUM_R_CORES                     : natural := 8;
        NUM_SEEDS                       : natural := 16;
        NUM_VOXELS                      : natural := 2
    );
    Port
    (
                                                                                    ----------------------------------------------------
        aclk                            : IN   STD_LOGIC;                           --! AXI Clock                          (Active High)
        axi_aresetn                     : IN   STD_LOGIC;                           --! AXI Reset                            (Active Low)
        system_reset                    : IN   STD_LOGIC;                           --! System Reset                        (Active High)
                                                                                    ----------------------------------------------------

        N_SAMPLES                       : IN   STD_LOGIC_VECTOR ( 11 downto 0 );
        R_PRIME                         : IN   STD_LOGIC_VECTOR ( 31 downto 0 );
        CURRENT_ADDR                    : OUT  STD_LOGIC_VECTOR ( 16 downto 0 );


        clk                             : OUT  STD_LOGIC;
        RESULTADDR                      : OUT  STD_LOGIC_VECTOR ( 12 downto 0 );
        VOXELCOUNT                      : OUT  STD_LOGIC_VECTOR ( 11 downto 0 );
        VOXELCOUNTACC                   : OUT  STD_LOGIC_VECTOR ( 11 downto 0 );
        WR                              : OUT  STD_LOGIC_VECTOR ( 0 downto 0 );
        RESULT0                         : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        RESULTVO                        : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        RESULTSD                        : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        VOXIDX                          : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        BRAMPOS0                        : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        SEED                            : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        VOXEL                           : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        MULRESULT                       : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        MULVALID                        : OUT  STD_LOGIC;
        LASTIN                          : OUT  STD_LOGIC;
        LASTOUT0                        : OUT  STD_LOGIC;
        LASTOUT1                        : OUT  STD_LOGIC;
        LASTOUT2                        : OUT  STD_LOGIC;
        LASTOUT3                        : OUT  STD_LOGIC;
        LASTOUT4                        : OUT  STD_LOGIC;
        A                               : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        B                               : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
                                                                                    ----------------------------------------------------
                                                                                    --  Voxel AXI Stream Interface
        M_AXIS_MM2S_VOXELS_tdata        : IN   STD_LOGIC_VECTOR((DATA_WIDTH_BITS * NUM_VOXELS) - 1 downto 0);
        M_AXIS_MM2S_VOXELS_tkeep        : IN   STD_LOGIC_VECTOR((DATA_WIDTH_BYTES * NUM_VOXELS) - 1 downto 0);
        M_AXIS_MM2S_VOXELS_tlast        : IN   STD_LOGIC;                           --! Data Last                           (Active High)
        M_AXIS_MM2S_VOXELS_tready       : OUT  STD_LOGIC;                           --! Interface Ready                     (Active High)
        M_AXIS_MM2S_VOXELS_tvalid       : IN   STD_LOGIC;                           --! Data Valid                          (Active High)
                                                                                    ----------------------------------------------------


                                                                                    ----------------------------------------------------
        SEEDS_BRAM_PORT_0_addr          : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );-- Seed BRAM Port
        SEEDS_BRAM_PORT_0_clk           : OUT  STD_LOGIC;                           --
        SEEDS_BRAM_PORT_0_din           : OUT  STD_LOGIC_VECTOR ( 511 downto 0 );
        SEEDS_BRAM_PORT_0_dout          : IN   STD_LOGIC_VECTOR ( 511 downto 0 );
        SEEDS_BRAM_PORT_0_en            : OUT  STD_LOGIC;                           --
        SEEDS_BRAM_PORT_0_rst           : OUT  STD_LOGIC;                           --
        SEEDS_BRAM_PORT_0_we            : OUT  STD_LOGIC_VECTOR ( 63 downto 0 );--
                                                                                    ----------------------------------------------------



                                                                                    ----------------------------------------------------
        RESULT_BRAM_PORT_0_addr         : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );-- Result BRAM Port
        RESULT_BRAM_PORT_0_clk          : OUT  STD_LOGIC;                           --
        RESULT_BRAM_PORT_0_din          : OUT  STD_LOGIC_VECTOR ( 63 downto 0 );--
        RESULT_BRAM_PORT_0_en           : OUT  STD_LOGIC;                           --
        RESULT_BRAM_PORT_0_rst          : OUT  STD_LOGIC;                           --
        RESULT_BRAM_PORT_0_we           : OUT  STD_LOGIC_VECTOR ( 7 downto 0 )      --
                                                                                    ----------------------------------------------------
    );
end cca_core;
```

Figure 4.10: HPCME NDE - VHDL: Ports and Core Configuration Parameters

This VHDL section listed in Figure 4.10 provides the interface to the rest of the NDE hardware. There is a voxel stream port (AXI interface), a seed BRAM port, and a result BRAM port. There are also register ports to set the number of samples, the correlation threshold, and observe the current result address to know when to transfer results from the BRAM to the SDRAM and finally

to FLASH storage. Also, there are various diagnostic ILA ports to enable developers to verify the internal states of the VVCA core.

This VHDL was written to enable a flexible seed / voxel structure. The configuration parameters set the number of voxels and seeds that are to be processed simultaneously. The default configuration is a 2 voxel by 16 seed, however, the configuration may easily be changed to 4 voxel by 32 seeds with just modifying the configuration parameters. The VHDL will reconfigure the bus structures to enable this change. However, the underlying FPGA implementation must be able to support the number of instances of floating point computation cores that will be generated.

```
------------------------------------------------------------------------------------------------------------------------
-- Components
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
-- Floating Point Multiplier Component
-- Result = A * B
------------------------------------------------------------------------------------------------------------------------
component floating_point_mul is
port
(
    aclk                        : IN  STD_LOGIC;                    --! Clock                          (Active High)
    s_axis_a_tvalid             : IN  STD_LOGIC;                    --! Port "A" Data Valid            (Active High)
    s_axis_a_tready             : OUT STD_LOGIC;                    --! Port "A" Data Ready            (Active High)
    s_axis_a_tdata              : IN  STD_LOGIC_VECTOR(31 DOWNTO 0); --! Port "A" Data                   (32-bit Bus)
    s_axis_a_tlast              : IN  STD_LOGIC;                    --! Port "A" Data Last             (Active High)
    s_axis_b_tvalid             : IN  STD_LOGIC;                    --! Port "B" Data Valid            (Active High)
    s_axis_b_tready             : OUT STD_LOGIC;                    --! Port "B" Data Ready            (Active High)
    s_axis_b_tdata              : IN  STD_LOGIC_VECTOR(31 DOWNTO 0); --! Port "B" Data                   (32-bit Bus)
    s_axis_b_tlast              : IN  STD_LOGIC;                    --! Port "B" Data Last             (32-bit Bus)
    m_axis_result_tvalid        : OUT STD_LOGIC;                    --! Result Valid                   (Active High)
    m_axis_result_tready        : IN  STD_LOGIC;                    --! Result Ready                   (Active High)
    m_axis_result_tdata         : OUT STD_LOGIC_VECTOR(31 DOWNTO 0); --! Result Data                   (32-bit Bus)
    m_axis_result_tlast         : OUT STD_LOGIC                     --! Result Last                   (Active High)
);
end component;


------------------------------------------------------------------------------------------------------------------------
-- Floating Point Accumulator Component
-- Result = Result + A
------------------------------------------------------------------------------------------------------------------------
component floating_point_acc is
port
(
    aclk                        : IN  STD_LOGIC;                    --! Clock                          (Active High)
    aresetn                     : IN  STD_LOGIC;                    --! AReset                         (Active Low)
    s_axis_a_tvalid             : IN  STD_LOGIC;                    --! Port "A" Data Valid            (Active High)
    s_axis_a_tready             : OUT STD_LOGIC;                    --! Port "A" Data Ready            (Active High)
    s_axis_a_tdata              : IN  STD_LOGIC_VECTOR(31 DOWNTO 0); --! Port "A" Data                   (32-bit Bus)
    s_axis_a_tlast              : IN  STD_LOGIC;                    --! Port "A" Last                  (Active High)
    m_axis_result_tvalid        : OUT STD_LOGIC;                    --! Result Valid                   (Active High)
    m_axis_result_tready        : IN  STD_LOGIC;                    --! Result Ready                   (Active High)
    m_axis_result_tdata         : OUT STD_LOGIC_VECTOR(31 DOWNTO 0); --! Result Data                   (32-bit Bus)
    m_axis_result_tlast         : OUT STD_LOGIC                     --! Result Last                   (Active High)
);
end component;
```

Figure 4.11: HPCME NDE - VHDL: Floating Point Multiplier and Accumulator

```
-------------------------------------------------------------------------------------------------------------------------
-- Floating Point Divider Component
-- Result = A / B
-------------------------------------------------------------------------------------------------------------------------
component floating_point_div is
port
(
    aclk                        : IN   STD_LOGIC;                       --! Clock                          (Active High)
    s_axis_a_tvalid             : IN   STD_LOGIC;                       --! Port "A" Data Valid             (Active High)
    s_axis_a_tready             : OUT  STD_LOGIC;                       --! Port "A" Data Ready             (Active High)
    s_axis_a_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "A" Data                     (32-bit Bus)
    s_axis_b_tvalid             : IN   STD_LOGIC;                       --! Port "B" Data Valid             (Active High)
    s_axis_b_tready             : OUT  STD_LOGIC;                       --! Port "B" Data Ready             (Active High)
    s_axis_b_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "B" Data                     (32-bit Bus)
    m_axis_result_tvalid        : OUT  STD_LOGIC;                       --! Result Valid                    (Active High)
    m_axis_result_tready        : IN   STD_LOGIC;                       --! Result Ready                    (Active High)
    m_axis_result_tdata         : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0)    --! Result Data                     (32-bit Bus)
);
end component;

-------------------------------------------------------------------------------------------------------------------------
-- Floating Point Compare Component
-- Result = A >= B
-------------------------------------------------------------------------------------------------------------------------
component floating_point_comp is
port
(
    aclk                        : IN   STD_LOGIC;                       --! Clock                          (Active High)
    s_axis_a_tvalid             : IN   STD_LOGIC;                       --! Port "A" Data Valid             (Active High)
    s_axis_a_tready             : OUT  STD_LOGIC;                       --! Port "A" Data Ready             (Active High)
    s_axis_a_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "A" Data                     (32-bit Bus)
    s_axis_b_tvalid             : IN   STD_LOGIC;                       --! Port "B" Data Valid             (Active High)
    s_axis_b_tready             : OUT  STD_LOGIC;                       --! Port "B" Data Ready             (Active High)
    s_axis_b_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "B" Data                     (32-bit Bus)
    m_axis_result_tvalid        : OUT  STD_LOGIC;                       --! Result Valid                    (Active High)
    m_axis_result_tready        : IN   STD_LOGIC;                       --! Result Ready                    (Active High)
    m_axis_result_tdata         : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)     --! Result Data                     (32-bit Bus)
);
end component;

    -------------------------------------------------------------------------------------------------------------------------
-- Floating Point Compare Component
-- Result = A <= B
-------------------------------------------------------------------------------------------------------------------------
component floating_point_complte is
port
(
    aclk                        : IN   STD_LOGIC;                       --! Clock                          (Active High)
    s_axis_a_tvalid             : IN   STD_LOGIC;                       --! Port "A" Data Valid             (Active High)
    s_axis_a_tready             : OUT  STD_LOGIC;                       --! Port "A" Data Ready             (Active High)
    s_axis_a_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "A" Data                     (32-bit Bus)
    s_axis_b_tvalid             : IN   STD_LOGIC;                       --! Port "B" Data Valid             (Active High)
    s_axis_b_tready             : OUT  STD_LOGIC;                       --! Port "B" Data Ready             (Active High)
    s_axis_b_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "B" Data                     (32-bit Bus)
    m_axis_result_tvalid        : OUT  STD_LOGIC;                       --! Result Valid                    (Active High)
    m_axis_result_tready        : IN   STD_LOGIC;                       --! Result Ready                    (Active High)
    m_axis_result_tdata         : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)     --! Result Data                     (32-bit Bus)
);
end component;
```

Figure 4.12: HPCME NDE - VHDL: Floating Point Divider and Comparators

This VHDL section list in Figures 4.11 and 4.12 show the I/O configuration of the LogiCORE floating point modules that are used for this core. The IP for this module has been configured to utilize the built in DSP resources, support blocking mode, and support max latency. As we are concerned more with continual streaming of data than the latency of the computation this is ideal and will consume less resources. The blocking mode is configured to align the inputs to the various cores such that the core waits until both inputs are valid before allowing a computation to process. This is required to synchronize the calculations for the PCC.

```
-------------------------------------------------------------------------------------------------------------------------
-- Block Memory State Machine
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-- Process:         BRAM Data
-- Input(s):        aclk
-- Output(s):       BRAM Data In Port
-- Description:     Save results to BRAM
process (aclk)
begin
    if (falling_edge(aclk)) then
        case post_comp_valid(0) is
            when X"02" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(0);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(0);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(0);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(0);
                BRAMP0S0                                 <= r_value(0);
            when X"04" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(1);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(1);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(1);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(1);
                BRAMP0S0                                 <= r_value(1);
            when X"06" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(2);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(2);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(2);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(2);
                BRAMP0S0                                 <= r_value(2);
            when X"08" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(3);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(3);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(3);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(3);
                BRAMP0S0                                 <= r_value(3);
            when X"0A" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(4);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(4);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(4);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(4);
                BRAMP0S0                                 <= r_value(4);
            when X"0C" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(5);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(5);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(5);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(5);
                BRAMP0S0                                 <= r_value(5);
            when X"0E" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(6);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(6);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(6);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(6);
                BRAMP0S0                                 <= r_value(6);
            when X"10" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(7);
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(7);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(7);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(7);
                BRAMP0S0                                 <= r_value(7);
            when X"00" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= X"05555555";
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(0);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(0);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(0);
                BRAMP0S0                                 <= X"05555555";
            when X"12" =>
                RESULT_BRAM_PORT_0_din(31 downto 0)      <= X"55555555";
                RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(0);
                RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(0);
                RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(0);
                BRAMP0S0                                 <= X"55555555";
            when others =>
        end case;
    end if;
end process;
```

Figure 4.13: HPCME NDE - VHDL: Result BRAM data state machine

```
-------------------------------------------------------------------------------------------------------------------
-- Process:         BRAM Write Enable
-- Input(s):        aclk
-- Output(s):       BRAM Data WE Port
-- Description:     Save results to BRAM
process (aclk)
begin
    if (falling_edge(aclk)) then
        if (got_data(0) = '1') then
            case post_comp_valid(0) is
                when X"03" =>
                    if (r_capture(0) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when X"05" =>
                    if (r_capture(1) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when X"07" =>
                    if (r_capture(2) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when X"09" =>
                    if (r_capture(3) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when X"0B" =>
                    if (r_capture(4) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when X"0D" =>
                    if (r_capture(5) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when X"0F" =>
                    if (r_capture(6) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when X"11" =>
                    if (r_capture(7) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when others =>
                    RESULT_BRAM_PORT_0_we       <= X"00";
                    WR(0) <= '0';
            end case;
        else
            RESULT_BRAM_PORT_0_we       <= X"00";
            WR(0) <= '0';
        end if;
    end if;
end process;
```

Figure 4.14: HPCME NDE - VHDL: Result BRAM address state machine

```
-------------------------------------------------------------------------------------------------
-- Process:         BRAM Address Generator
-- Input(s):        aclk
-- Output(s):       BRAM Data WE Port
-- Description:     Save results to BRAM
process (aclk)
begin
    if (falling_edge(aclk)) then
        if axi_aresetn = '0' or system_reset = '1' or got_data(0) = '0' then
            result_address                      <= "0000000000000000";
        else
            case post_comp_valid(0) is
                when X"04" =>
                    if (r_capture(0) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"06" =>
                    if (r_capture(1) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"08" =>
                    if (r_capture(2) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"0A" =>
                    if (r_capture(3) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"0C" =>
                    if (r_capture(4) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"0E" =>
                    if (r_capture(5) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"10" =>
                    if (r_capture(6) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"12" =>
                    if (r_capture(7) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when others =>
            end case;
        end if;
    end if;

    RESULT_BRAM_PORT_0_addr(2 downto 0)   <= "000";
    RESULT_BRAM_PORT_0_addr(19 downto 3)  <= result_address;
    RESULT_BRAM_PORT_0_addr(31 downto 20) <= X"000";

    SEEDS_BRAM_PORT_0_addr(5 downto 0)  <= "000000";
    SEEDS_BRAM_PORT_0_addr(17 downto 6) <= seed_count;
    SEEDS_BRAM_PORT_0_addr(31 downto 18) <= "00000000000000";
end process;
```

Figure 4.15: HPCME NDE - VHDL: Result BRAM write enable state machine

The data port for the results is multiplexed to enable the use of only one 64-bit wide result port from eight correlation coefficient computations that occur simultaneously. The state machine to control this is shown in Figures 4.13, 4.14, and 4.15.

```
-------------------------------------------------------------------------------------------------------------------
-- Loop through all "R" cores
-------------------------------------------------------------------------------------------------------------------
RGEN : for r_index in 0 to (NUM_R_CORES - 1) generate
    ---------------------------------------------------------------------------------------------------------------
    -- Process:         "r" Divider State Machine
    -- Input(s):        aclk, Voxel Count
    -- Output(s):       BRAM Address
    -- Description:     Updates BRAM Address, Reset on start or increments on specific voxel count
    process (aclk, voxel_count)
    begin
        if (falling_edge(aclk)) then
            case voxel_count_acc(r_index) is
                when X"01E" =>
                    a_dividend_data(r_index)    <=  RESULTS_COV(r_index);
                    b_divisor_data(r_index)     <=  RESULTS_SD(r_index);
                    a_dividend_valid(r_index)   <= '1';
                    b_divisor_valid(r_index)    <= '1';
                when X"032" =>
                    a_dividend_data(r_index)    <=  RESULTS_COV(r_index + 8);
                    b_divisor_data(r_index)     <=  RESULTS_SD(r_index + 8);
                    a_dividend_valid(r_index)   <= '1';
                    b_divisor_valid(r_index)    <= '1';
                when X"046" =>
                    a_dividend_data(r_index)    <=  RESULTS_COV(r_index + 16);
                    b_divisor_data(r_index)     <=  RESULTS_SD(r_index + 16);
                    a_dividend_valid(r_index)   <= '1';
                    b_divisor_valid(r_index)    <= '1';
                when X"05A" =>
                    a_dividend_data(r_index)    <=  RESULTS_COV(r_index + 24);
                    b_divisor_data(r_index)     <=  RESULTS_SD(r_index + 24);
                    a_dividend_valid(r_index)   <= '1';
                    b_divisor_valid(r_index)    <= '1';
                when others =>
                    a_dividend_valid(r_index)   <= '0';
                    b_divisor_valid(r_index)    <= '0';
            end case;
        end if;
    end process;
```

Figure 4.16: HPCME NDE - VHDL: Correlation Coefficient Calculation state machine

```
--------------------------------------------------------------------------------------------------------------------------
--! U_DIV: floating_point_div
--! Clock on positive edge of ACLK
--! Perform division of Covar and SD Term
--------------------------------------------------------------------------------------------------------------------------
U_DIV : floating_point_div
Port Map
(
    aclk                        => aclk,
    s_axis_a_tvalid             => a_dividend_valid(r_index),
    s_axis_a_tready             => open,
    s_axis_a_tdata              => a_dividend_data(r_index),
    s_axis_b_tvalid             => b_divisor_valid(r_index),
    s_axis_b_tready             => open,
    s_axis_b_tdata              => b_divisor_data(r_index),
    m_axis_result_tvalid        => r_quotient_valid(r_index),
    m_axis_result_tready        => '1',
    m_axis_result_tdata         => r_quotient_data(r_index)
);

--------------------------------------------------------------------------------------------------------------------------
--! U_COMP: floating_point_comp
--! Clock on positive edge of ACLK
--! Perform comparison of "r" value ("A") to r' ("B") set to application
--------------------------------------------------------------------------------------------------------------------------
U_COMP : floating_point_comp
Port Map
(
    aclk                        => aclk,
    s_axis_a_tvalid             => r_quotient_valid(r_index),
    s_axis_a_tready             => open,
    s_axis_a_tdata              => r_quotient_data(r_index),
    s_axis_b_tvalid             => b_comp_valid,
    s_axis_b_tready             => open,
    s_axis_b_tdata              => b_comp_data,
    m_axis_result_tvalid        => r_comp_valid(r_index),
    m_axis_result_tready        => '1',
    m_axis_result_tdata         => r_comp_data(r_index)
);

--------------------------------------------------------------------------------------------------------------------------
--! U_COMPMAX: floating_point_comp
--! Clock on positive edge of ACLK
--! Perform comparison of "r" value ("A") to r' ("B") set to application (i.e. "r" <= 1.0)
--------------------------------------------------------------------------------------------------------------------------
U_COMPLTE : floating_point_complte
Port Map
(
    aclk                        => aclk,
    s_axis_a_tvalid             => r_quotient_valid(r_index),
    s_axis_a_tready             => open,
    s_axis_a_tdata              => r_quotient_data(r_index),
    s_axis_b_tvalid             => '1',
    s_axis_b_tready             => open,
    s_axis_b_tdata              => X"3F800000",
    m_axis_result_tvalid        => r_complte_valid(r_index),
    m_axis_result_tready        => '1',
    m_axis_result_tdata         => r_complte_data(r_index)
);
```

Figure 4.17: HPCME NDE - VHDL: Correlation Coefficient Calculation floating point assignments

```
    ----------------------------------------------------------------------------------------------------------
    -- Process:         Capture "r" value
    -- Input(s):        aclk
    -- Output(s):       Captured "r" data
    -- Description:     Save results to register
    process (aclk)
    begin
        if (rising_edge(aclk)) then
            if (r_quotient_valid(r_index) = '1') then
                r_value(r_index)            <= r_quotient_data(r_index);
            end if;
        end if;
    end process;

    ----------------------------------------------------------------------------------------------------------
    -- Process:         Capture "r" value index - R-Index (group)
    -- Input(s):        aclk
    -- Output(s):       Captured "r" data
    -- Description:     Save results to register
    process (aclk)
    begin
        if (rising_edge(aclk)) then
            if axi_aresetn = '0' or system_reset = '1' or accumulator_last_out(r_index) = '1' then
                r_value_grp_index(r_index)                  <= "11";
            elsif (r_quotient_valid(r_index) = '1') then
                r_value_grp_index(r_index)                  <= std_logic_vector(unsigned(r_value_grp_index(r_index)) + 1);
            end if;
        end if;
    end process;

    ----------------------------------------------------------------------------------------------------------
    -- Process:         Capture "r" value
    -- Input(s):        aclk
    -- Output(s):       Captured "r" data
    -- Description:     Save results to register
    process (aclk)
    begin
        if (rising_edge(aclk)) then
            if axi_aresetn = '0' or system_reset = '1' or got_data(0) = '0' then
                r_capture(r_index)  <= '0';
            elsif (r_comp_valid(r_index) = '1') and (r_complte_valid(r_index) = '1') then
                if (r_comp_data(r_index)(0) = '1') and (r_complte_data(r_index)(0) = '1') then
                    r_capture(r_index)  <= '1';
                else
                    r_capture(r_index)  <= '0';
                end if;
            end if;
        end if;
    end process;

    ----------------------------------------------------------------------------------------------------------
    -- Process:         Post valid clock counter
    -- Input(s):        aclk
    -- Output(s):       Captured "r" data
    -- Description:     Save results to register
    process (aclk)
    begin
        if (rising_edge(aclk)) then
            if (r_comp_valid(r_index) = '1') then
                post_comp_valid(r_index)    <= X"00";
            elsif (post_comp_valid(r_index) < X"80") then
                post_comp_valid(r_index)    <= std_logic_vector(unsigned(post_comp_valid(r_index)) + 1);
            end if;
        end if;
    end process;

end generate RGEN;
```

Figure 4.18: HPCME NDE - VHDL: Correlation Coefficient Calculation result capture

Based on the number of result processing cores assigned, in this implementation, the VHDL generates the logic for each core. A state machine, shown in Figure 4.16, selects which internal register to assign to the floating point modules to coordinate a 4-to-1 data multiplexer. The floating point cores are assigned as shown in Figure 4.17. The correlation results are then computed and

then compared using the two comparators, which is shown in Figure 4.18. These two comparators, as discussed previously, verify a valid correlation coefficient and determines if the threshold has been met. If these conditions are met then the data is flagged to be captured by the previously discussed results memory state machine.

```
-------------------------------------------------------------------------------------------------------------------
-- Loop through all seeds and voxels
-------------------------------------------------------------------------------------------------------------------
SEEDGEN : for seed_index in 0 to (NUM_SEEDS - 1) generate
    VOXELGEN : for voxel_index in 0 to (NUM_VOXELS - 1) generate
        -------------------------------------------------------------------------------------------------------------------
        --! U_MUL: floating_point_mult
        --! Clock on positive edge of ACLK
        --! Perform multiplication of Seed and Voxel
        -------------------------------------------------------------------------------------------------------------------
        U_MUL : floating_point_mul
        Port Map
        (
            aclk                        => aclk,
            s_axis_a_tvalid             => REG_VOXELS_tvalid,
            s_axis_a_tready             => M_AXIS_MM2S_VOXELS_ready((seed_index * NUM_VOXELS) + voxel_index),
            s_axis_a_tdata              => REG_VOXELS_tdata(((voxel_index * DATA_WIDTH_BITS) + 31)  downto (voxel_index * DATA_WIDTH_BITS)),
            s_axis_a_tlast              => REG_VOXELS_tlast,
            s_axis_b_tvalid             => '1',
            s_axis_b_tready             => open,
            s_axis_b_tdata              => SEEDS_BRAM_PORT_0_dout(((seed_index * DATA_WIDTH_BITS) + 31)  downto (seed_index * DATA_WIDTH_BITS)),
            s_axis_b_tlast              => '0',
            m_axis_result_tvalid        => multiplier_valid((seed_index * NUM_VOXELS) + voxel_index),
            m_axis_result_tready        => accumulator_ready((seed_index * NUM_VOXELS) + voxel_index),
            m_axis_result_tdata         => multiplier_result((seed_index * NUM_VOXELS) + voxel_index),
            m_axis_result_tlast         => multiplier_last((seed_index * NUM_VOXELS) + voxel_index)
        );

        -------------------------------------------------------------------------------------------------------------------
        --! U_ACC: floating_point_acc
        --! Clock on positive edge of ACLK
        --! Perform accumulation of Seed and Voxel
        -------------------------------------------------------------------------------------------------------------------
        U_ACC : floating_point_acc
            Port Map
            (
                aclk                        => aclk,
                aresetn                     => accumulator_reset((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tvalid             => multiplier_valid((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tready             => accumulator_ready((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tdata              => multiplier_result((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tlast              => accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index),
                m_axis_result_tvalid        => accumulator_valid((seed_index * NUM_VOXELS) + voxel_index),
                m_axis_result_tready        => '1',
                m_axis_result_tdata         => RESULTS_0((seed_index * NUM_VOXELS) + voxel_index),
                m_axis_result_tlast         => accumulator_last_out((seed_index * NUM_VOXELS) + voxel_index)
            );
```

Figure 4.19: HPCME NDE - VHDL: Covariance Core - Multiply and Accumulators

In this section of code shown in Figure 4.19, the actual covariance cores are generated. This section uses one floating point multiplier and accumulator per seed-voxel pair. The multiplier uses one AXI streaming port for the voxels and the seeds are read from a high-speed dual port BRAM. The seed BRAM is always in read mode and has the address generator state machine defined in a similar way as the result address.

```
-- Capture results - Result Bank Voxel #0 or Voxel #1
process (aclk, accumulator_last_out)
begin
    if (rising_edge(aclk)) then
        if (accumulator_last_out_1((seed_index * NUM_VOXELS) + voxel_index) = '1' and
            accumulator_last_out_2((seed_index * NUM_VOXELS) + voxel_index) = '0' and
            accumulator_last_out_3((seed_index * NUM_VOXELS) + voxel_index) = '0') then
            RESULTS_COV((seed_index * NUM_VOXELS) + voxel_index)    <= RESULTS_0((seed_index * NUM_VOXELS) + voxel_index);
        end if;
    end if;
end process;


process (aclk)
begin
    if (rising_edge(aclk)) then
        sample_count_1((seed_index * NUM_VOXELS) + voxel_index)          <=  std_logic_vector(unsigned(N_SAMPLES) + 1);
        sample_count_2((seed_index * NUM_VOXELS) + voxel_index)          <=  std_logic_vector(unsigned(N_SAMPLES) + 2);
        sample_count_3((seed_index * NUM_VOXELS) + voxel_index)          <=  std_logic_vector(unsigned(N_SAMPLES) - 4);
    end if;
end process;

-------------------------------------------------------------------------------------------------
-- Process:         Last Out Delay Registers
-- Input(s):        aclk
-- Output(s):       Last Outs
-- Description:     4-stage last out register
process (aclk, accumulator_last_out, accumulator_last_out_1, accumulator_last_out_2, accumulator_last_out_3)
begin
    if (falling_edge(aclk)) then
        accumulator_last_out_1((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out((seed_index * NUM_VOXELS) + voxel_index);
        accumulator_last_out_2((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out_1((seed_index * NUM_VOXELS) + voxel_index);
        accumulator_last_out_3((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out_2((seed_index * NUM_VOXELS) + voxel_index);
        accumulator_last_out_4((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out_3((seed_index * NUM_VOXELS) + voxel_index);
    end if;
end process;
```

Figure 4.20: HPCME NDE - VHDL: Covariance Core - Covariance and Standard Deviation capture registers

In this section of code shown in Figure 4.20, when N samples have been accumulated (last out is set) the covariance is captured. Here other sample counter and holding registers are set for internal logic.

```
-------------------------------------------------------------------------------------------------
-- Process:         Internal Voxel Data Counter and Accumulator Last Signal - Multiplier Valid
-- Input(s):        aclk
-- Output(s):       Voxel Counter
-- Description:     Counts voxels until reset
process (aclk, voxel_count)
begin
    if (rising_edge(aclk)) then
        -- If voxel count greater than spec. value then reset
        if  (axi_aresetn = '0' or system_reset = '1') then
            accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '0';-- CHANGE from zero
            got_data((seed_index * NUM_VOXELS) + voxel_index) <= '0';
            voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= X"000";
        elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) = N_SAMPLES) and
              (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
            accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '1';
            got_data((seed_index * NUM_VOXELS) + voxel_index) <= '1';
            voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
        elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) = sample_count_1((seed_index * NUM_VOXELS) + voxel_index)) and
              (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
            accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '1';
            voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
        elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) = sample_count_2((seed_index * NUM_VOXELS) + voxel_index)) and
              (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
            accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '1';
            got_data((seed_index * NUM_VOXELS) + voxel_index) <= '1';
            voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
            RESULTS_SD((seed_index * NUM_VOXELS) + voxel_index)          <= multiplier_result((seed_index * NUM_VOXELS) + voxel_index);
        elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) > N_SAMPLES) and (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
            accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '0';
            voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= X"000";
        elsif multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1' then
            accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '0';
            voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
        end if;
    end if;
end process;


-------------------------------------------------------------------------------------------------
-- Process:         Accumulator Reset
-- Input(s):        aclk
-- Output(s):       Reset
-- Description:      Accumulator Reset
process (aclk, voxel_count)
begin
    if (rising_edge(aclk)) then
        if  (axi_aresetn = '0' or system_reset = '1') then
            accumulator_reset((seed_index * NUM_VOXELS) + voxel_index) <= '0';
        else
            accumulator_reset((seed_index * NUM_VOXELS) + voxel_index) <= '1';
        end if;
    end if;
end process;
```

Figure 4.21: HPCME NDE - VHDL: Covariance Core - Accumulator reset and counter control state machine 1 of 2

```
-------------------------------------------------------------------------------------------------
-- Process:         Internal Voxel Data Counter and Accumulator Last Signal - Accumulator Valid
-- Input(s):        aclk
-- Output(s):       Voxel Counter
-- Description:     Counts voxels until reset
process (aclk, voxel_count_acc)
begin
    if (rising_edge(aclk)) then
        -- If voxel count greater than spec. value then reset
        if  (axi_aresetn = '0' or system_reset = '1') then
            voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= X"000";
        elsif ((voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) = N_SAMPLES) or
               (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) = sample_count_1((seed_index * NUM_VOXELS) + voxel_index)) or
               (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) = sample_count_2((seed_index * NUM_VOXELS) + voxel_index))) and
              (accumulator_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
            voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= std_logic_vector(unsigned(voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)) + 1);
        elsif (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) > N_SAMPLES) and (accumulator_last_out_4((seed_index * NUM_VOXELS) + voxel_index) = '1') then
            voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= X"000";
        elsif (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) < X"FFF") and (got_data((seed_index * NUM_VOXELS) + voxel_index) = '1') then
            voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= std_logic_vector(unsigned(voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)) + 1);
        end if;
    end if;
end process;
```

Figure 4.22: HPCME NDE - VHDL: Covariance Core - Accumulator reset and counter control state machine 2 of 2

In this section of code shown in Figures 4.21 and 4.22, when N samples have been accumulated, based upon the voxel sample counter, a flag is set to force the covariance capture, discussed

previously, as to capture the embedded standard deviation of the voxel and seeds. This data is stored in registers for use in the previously discussed division step.

```
        -- For the first index set all bits to initial index value
        I0: if ((seed_index * NUM_VOXELS) + voxel_index) = 0 generate
            M_AXIS_MM2S_VOXELS_tready_T(0)  <= M_AXIS_MM2S_VOXELS_ready(0);
        end generate I0;

        -- Any other index and/or with the previous value
        IX: if ((seed_index * NUM_VOXELS) + voxel_index) > 0 generate
            M_AXIS_MM2S_VOXELS_tready_T((seed_index * NUM_VOXELS) + voxel_index) <= M_AXIS_MM2S_VOXELS_tready_T(((seed_index * NUM_VOXELS) + voxel_index) - 1)
                                                                                   and M_AXIS_MM2S_VOXELS_ready((seed_index * NUM_VOXELS) + voxel_index);
        end generate IX;
    end generate VOXELGEN;
end generate SEEDGEN;

-------------------------------------------------------------------------------------------------------------------
-- And results for output (i.e. get last index from gen)
-------------------------------------------------------------------------------------------------------------------
M_AXIS_MM2S_VOXELS_tready          <=  M_AXIS_MM2S_VOXELS_tready_T(((NUM_VOXELS * NUM_SEEDS)-1));
end RTL;
```

Figure 4.23: HPCME NDE - VHDL: Covariance Core - AXI data stream synchronization

In this final step, shown in Figures 4.23 the incoming voxel stream is synchronized to all of the various multipliers. This logic prevents any group of multiplication to finish first and cause a race condition which would result in incorrect results and data being unsynchronized.

The firmware source code is listed in the Source Code Appendix of this dissertation. The essential functions of the firmware are to:

- Initialize SDRAM memory regions (Voxel Memory, Results Vector, etc...)
- Initialize BRAM memory regions (Seed Memory, Results Vectors, etc...)
- Established communications with host
- Start processing task files loaded on SD Card
- Each task has a voxel group, seed group, and a voxel offset
- Load voxels and seeds into SDRAM
- Transfer seeds to BRAM, if same seed group as previous then this step is skipped
- Perform high-speed DMA transfer from SDRAM to the VVCA core
- Retrieve results from BRAM, store in SDRAM, and then transfer to the SD Card

- Once all groups are processed, wait for additional task instructions

Overall the firmware oversees the VVCA core process and primarily just moves data from the external storage media (SD Card) to either the SDRAM or BRAMs. Processing performance is achieved by having a significant amount of voxels stored in high-speed DDR3 SDRAM memory for efficient transfer via the DMA. This is further enhanced by having all of the seeds stored in a BRAM configured with a wide memory port. In this particular implementation the port was 512 bits wide. However, as was tested in the PoC system the port can be 1,024 bits wide or more.

It would also be possible to configure an implementation to use multiple VVCA cores if the FPGA has sufficient resources. For instance, this would be possible with a Xilinx Zynq 7Z045 or 7Z100 or the Xilinx UltraScale.

4.3 Initial HCP Test and Discussion

The HPCME system was initially tested using a HCP dataset provided the Human Connectome Project. This dataset has a resolution of 91 voxels in the coronal plane, 109 voxels in the sagittal plane, and 91 voxels in the transverse plane. This resolution provides a dataset of 902,629 voxels with a sample size of 1,200 samples.

This dataset was preprocessed and organized as described in the preceding preprocessing section and the resulting files copied to a FLASH SD card for utilization by the HPCME system. Task files were also generated for each NDE in the system, and with the optimization that each NDE in the system have ¼ of the task load.

After the files were loaded into the HPCME, an application running on a PC workstation laptop was activated, and the power applied to the HPCME. Communications were established with the HPCME, and the task processing system was initiated. These results, along with the others will be shown in the following section of this dissertation.

The HPCME was also tested using a logic analyzer and following each stage of the computation cycle in the correlation core. This logic analyzer enabled correction to the fixed point accumulator definition to be modified and corrected to result in the most optimum use of FPGA resources as well as accuracy of results.

The results of the correlation coefficients of the HCP dataset was verified by comparing correlation coefficients computed using a MATLAB script. This script loaded the computed results from the HPCME and for the same seed and voxel indicated computed a double precision correlation coefficient of seed-voxel pair as being connected. These tests illustrated that the HPCME mathematically matched the MATLAB results with negligible error (less than 0.00001%).

Further test results will be discussed in the following section.

5. VALIDATION OF HPCME SYSTEM

5.1 Introduction

Validation of the HPCME is focused on both the computational accuracy of the results of the HPCME as well as comparing the performance to various other processing methods. These methods include the HPCME, traditional CPU processing, a GPU implementation, and the CONN toolkit. It should be noted that the accuracy of the results by other methods was not the focus of this validation.

Dr. Orr, Department of Psychological and Brain Sciences at the Texas A&M Institute for Neuroscience, provided twenty-four HCP datasets that are being used in his ongoing research to be used to generate results using the HPCME. These HCP datasets were from the S900 Release of Human Connectome Project (WU-UMN HCP Consortium), whose purpose is to "recruit a sample of relatively healthy individuals free of a prior history of significant psychiatric or neurological illnesses. The goal was to capture a broad range of variability in healthy individuals concerning behavioral, ethnic, and socioeconomic diversity" [3].

Detailed descriptions of each variable used to eliminate participants are available here: https://wiki.humanconnectome.org (see HCP Data Dictionary Public - 500 Subject Release).

Van Essen and colleagues reported details on the data acquisition in the HCP sample [3]. rs-fMRI data for each participant consisted of 2, 15-minute runs (902,629 voxels, 1200 volumes, 720 ms TR, 2 mm isotropic voxels). Datasets were downloaded from the HCP S900 Release Resting-State fMRI 1 FIX-Denoised (Extended) Package which included preprocessed data that had been registered and denoised using the FIX ICA-based automated method. Additional details on this pipeline are discussed in detail elsewhere [11].

The CPU methods include a simplistic implementation of a correlation algorithm as well as results gathered from various other research articles with regards to building brain connectivity maps. The focus of the CPU comparisons was to illustrate the best case scenario that can be

substantiated through peer review literature and not just a single test case.

The GPU comparison also consisted of using a recently published correlation algorithm purposely built for processing HCP datasets. Results were collected and is shown and discussed in comparison to the performance of the HPCME.

Dr. Orr performed data analysis using the CONN toolkit and these results as discussed. Since the nature of processing high-resolution and high sample HCP datasets is not typically done, due to the lengthy computation time, comparisons were mostly focused on overlapping results and understanding the differences between the CONN toolkit and HPCME connectedness maps.

5.2 Results of HPCME Processing HCP Datasets

The results that have been generated in the testing of the HPCME demonstrate that the overall architecture and implementation as previously discussed do appear to provide full processing of HCP datasets within seven hours.

The HPCME system has been validated using datasets provided by Dr. Orr of the Texas A&M Brain Institute and the Human Connectome Project. This group of datasets consists of 24 subjects each with similar resolution and sample length to get a consistent comparison of performance and data validity. The resolution of each dataset is 91 voxels in the coronal plane, 109 voxels in the sagittal plane, and 91 voxels in the transverse plane. This resolution provides a dataset of 902,629 voxels with a sample size of 1,200 samples.

Each dataset was preprocessed and organized as described in the preceding preprocessing section and the resulting files copied to a FLASH SD card for utilization by the HPCME system. Task files were also generated for each NDE in the system, and with the optimization that each NDE in the system have 1/4 of the task load.

After the files were loaded into the HPCME, an application running on a PC workstation laptop was activated, and the power applied to the HPCME. Communications were established with the HPCME, and the task processing system was initiated.

It should be noted, that logging and monitoring of the results are not required during task processing of the data, as the HPCME does not require operator intervention in processing. However, for this specific application, all datasets being processed were monitored and the corresponding processing times recorded along with the final result vector files.

As a whole, the results do show that the average time to process the data was approximately 6½ hours, with a standard deviation of about 2½ minutes, or 2.4%. The average r-vector result file ranged in size from 26 MB to 928 MB when using a correlation threshold of 0.7. Another set of tests processed results using a correlation threshold of 0.63. The results of changing the correlation threshold are highlighted in the following charts, Figures 5.1 and 5.2, and Table 5.1.

By changing the correlation threshold the overall processing time did not change, to any degree.

However, the size of the resulting datasets did change from an average size of about 93 MB to about 4 GB. The data size increased due to the increase in correlation connectivity. It should be noted, that even though the data sizes are still significant, however, for such high connectivity this is still significantly less than the four terabytes of data which would be the result if no r-vectors were utilized as a traditional correlation matrix.

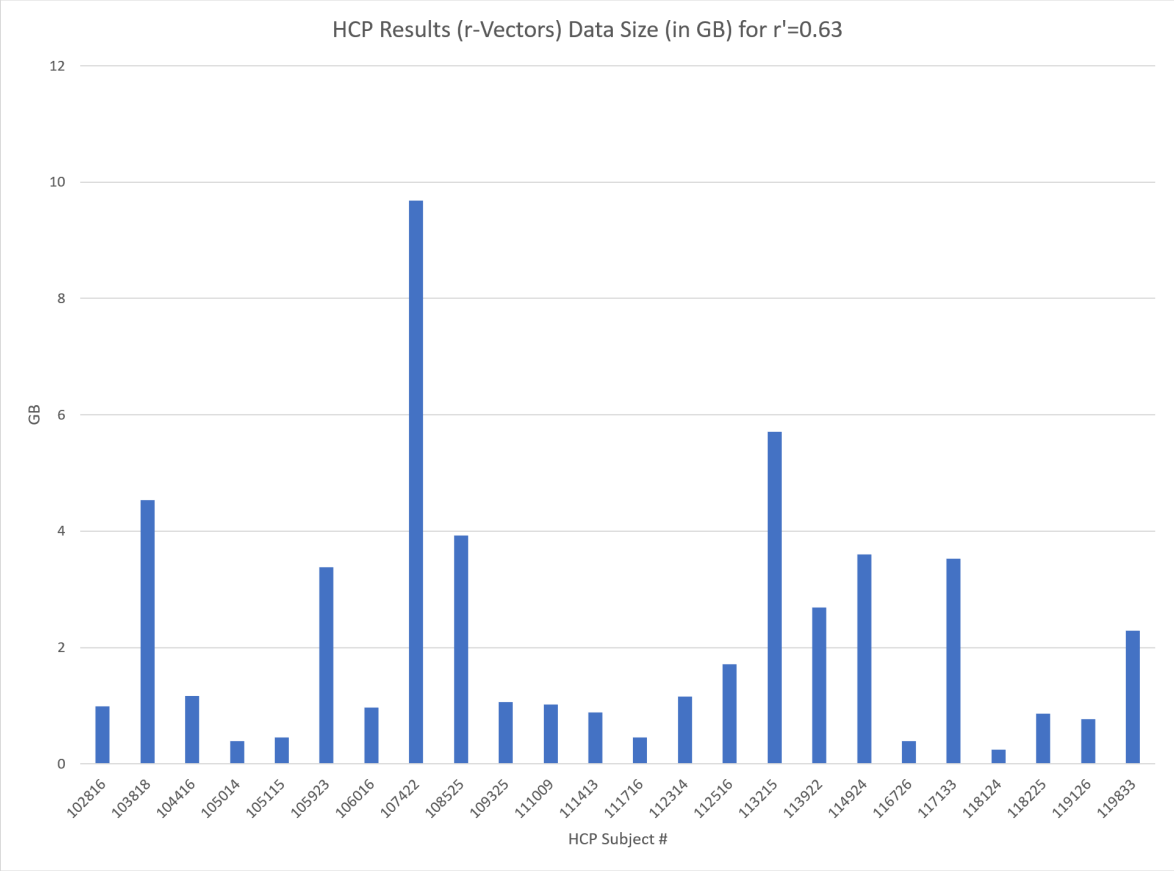


Figure 5.1: Results file size for HCP datasets with r=0.63

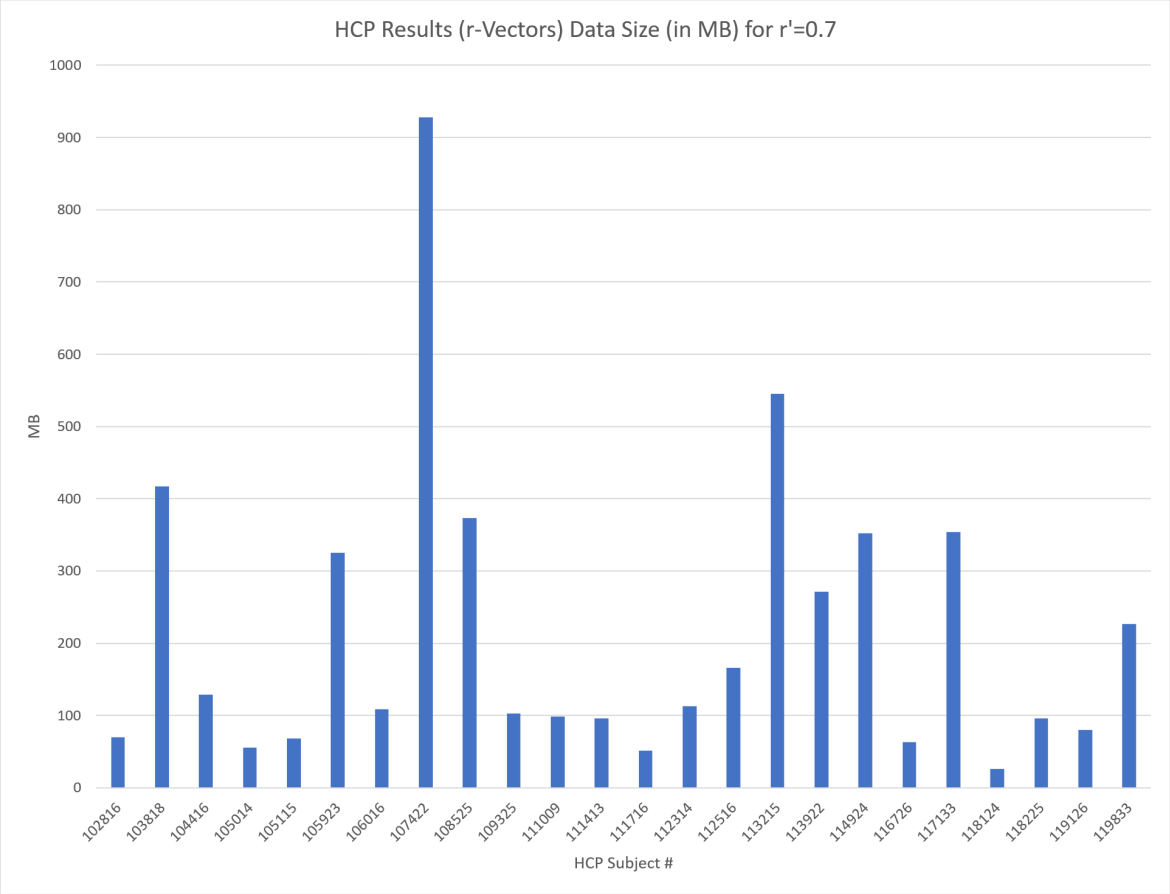


Figure 5.2: Results file size for HCP datasets with r=0.70

HCP Subject #:	r'=0.63		r='0.70	
102816	0.99	GB	70	MB
103818	4.53	GB	417	MB
104416	1.16	GB	129	MB
105014	0.39	GB	55.2	MB
105115	0.46	GB	68.5	MB
105923	3.38	GB	325	MB
106016	0.97	GB	109	MB
107422	9.68	GB	928	MB
108525	3.92	GB	373	MB
109325	1.06	GB	103	MB
111009	1.02	GB	98.3	MB
111413	0.88	GB	96.2	MB
111716	0.45	GB	51.6	MB
112314	1.16	GB	113	MB
112516	1.71	GB	166	MB
113215	5.71	GB	545	MB
113922	2.69	GB	271	MB
114924	3.60	GB	352	MB
116726	0.39	GB	63.2	MB
117133	3.53	GB	354	MB
118124	0.25	GB	26	MB
118225	0.87	GB	95.7	MB
119126	0.77	GB	80.2	MB
119833	2.29	GB	227	MB
MEAN:	2.16	GB	213.2077	MB

Table 5.1: Results file size for HCP datasets comparison

Once the data set had been collected for all 24 datasets, each result set was individually post-processed to generate a set of connectivity maps or connectedness maps as well as the statistical results on the time the processing system took to process each dataset.

In the following pages are a two representative results of the 24 resulting HCP datasets along with the processing times of all 24 datasets. These charts and connectedness maps demonstrate the ability to build connectedness maps rapidly utilizing the HPCME system. Connectedness maps of all 24 HCP datasets are shown in Appendix A.

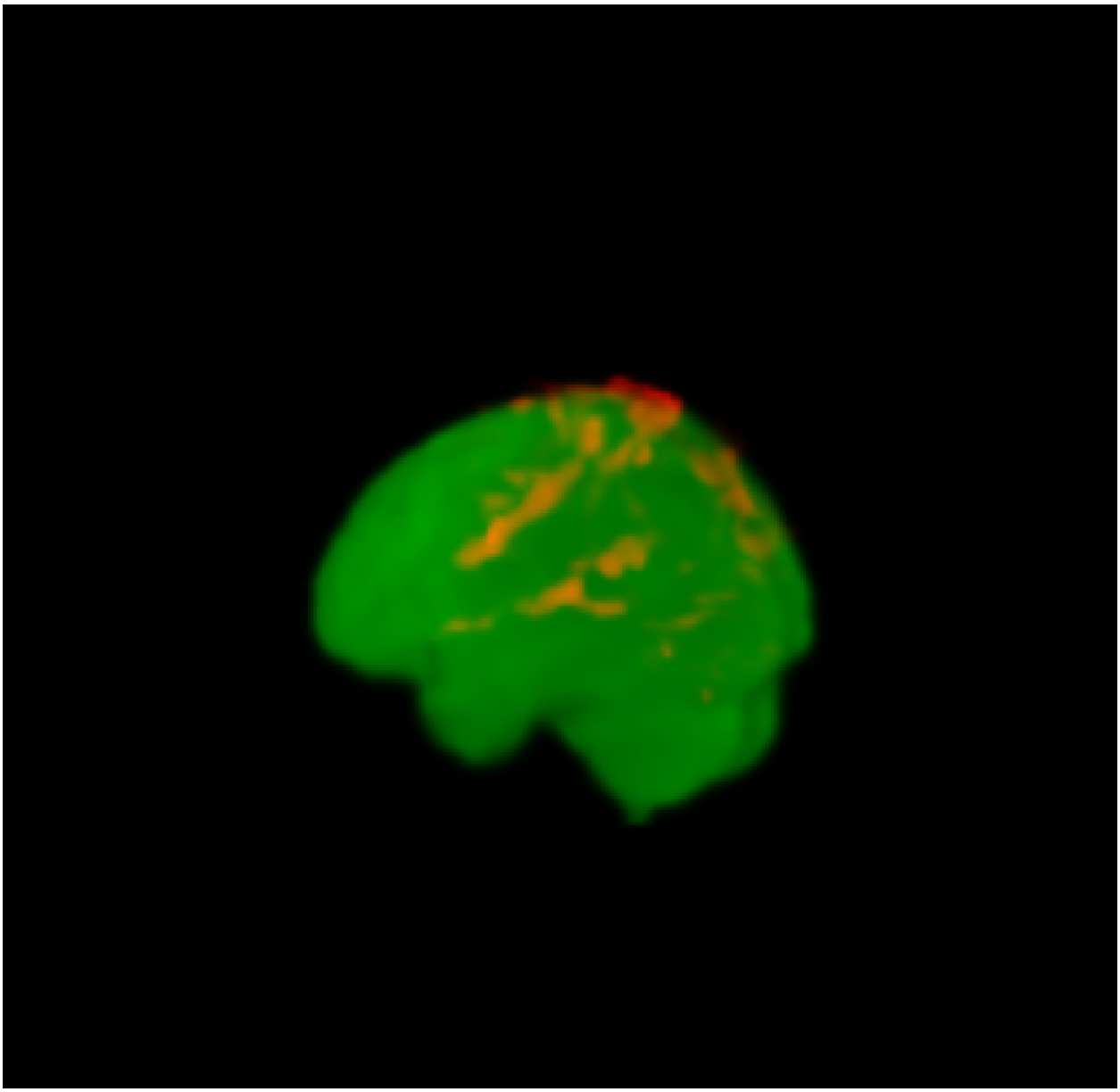


Figure 5.3: 3D representation of HCP Subject 102816 with r=0.63

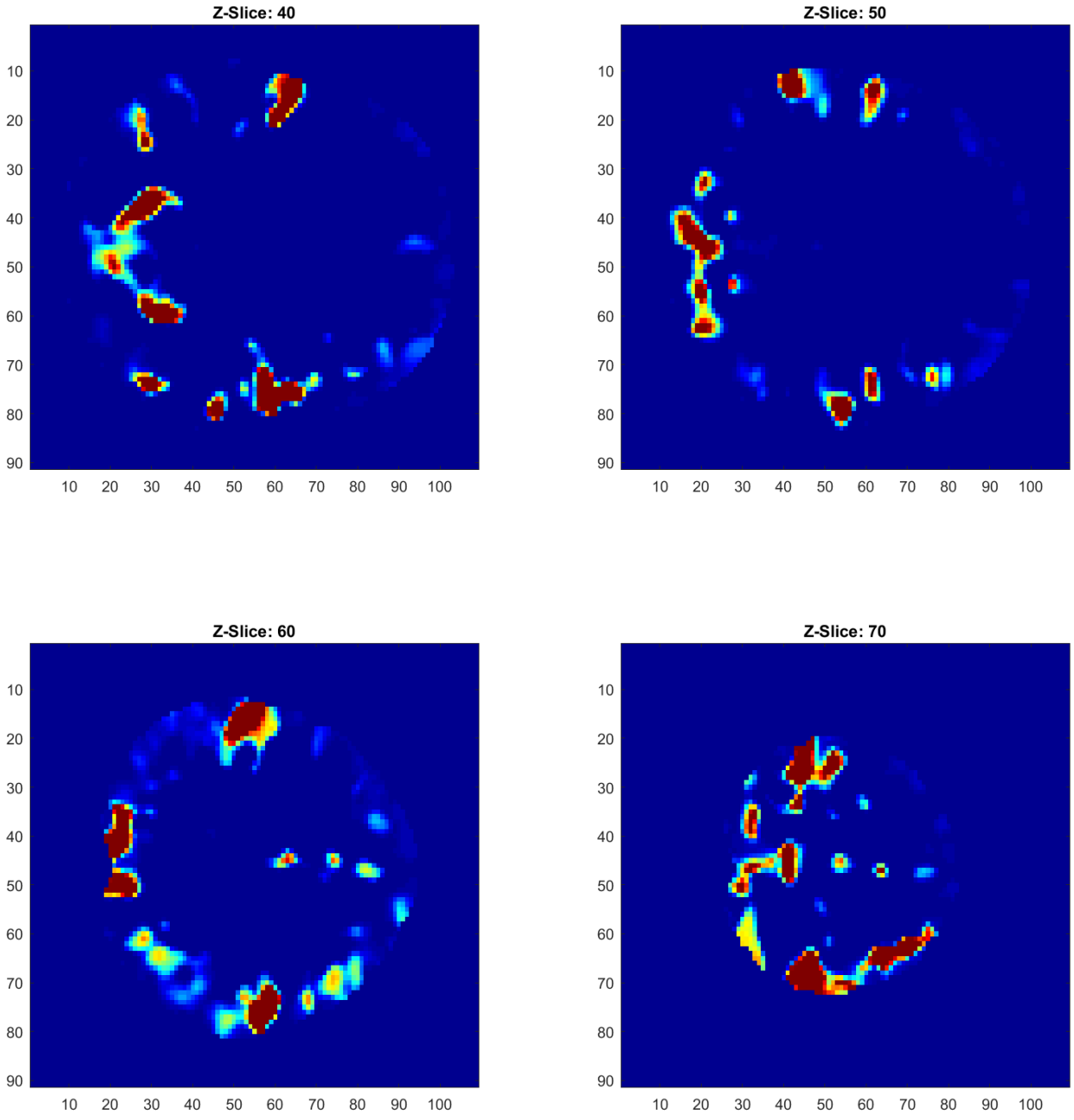


Figure 5.4: Connectedness of HCP Subject 102816 with r=0.63 (4-slices)

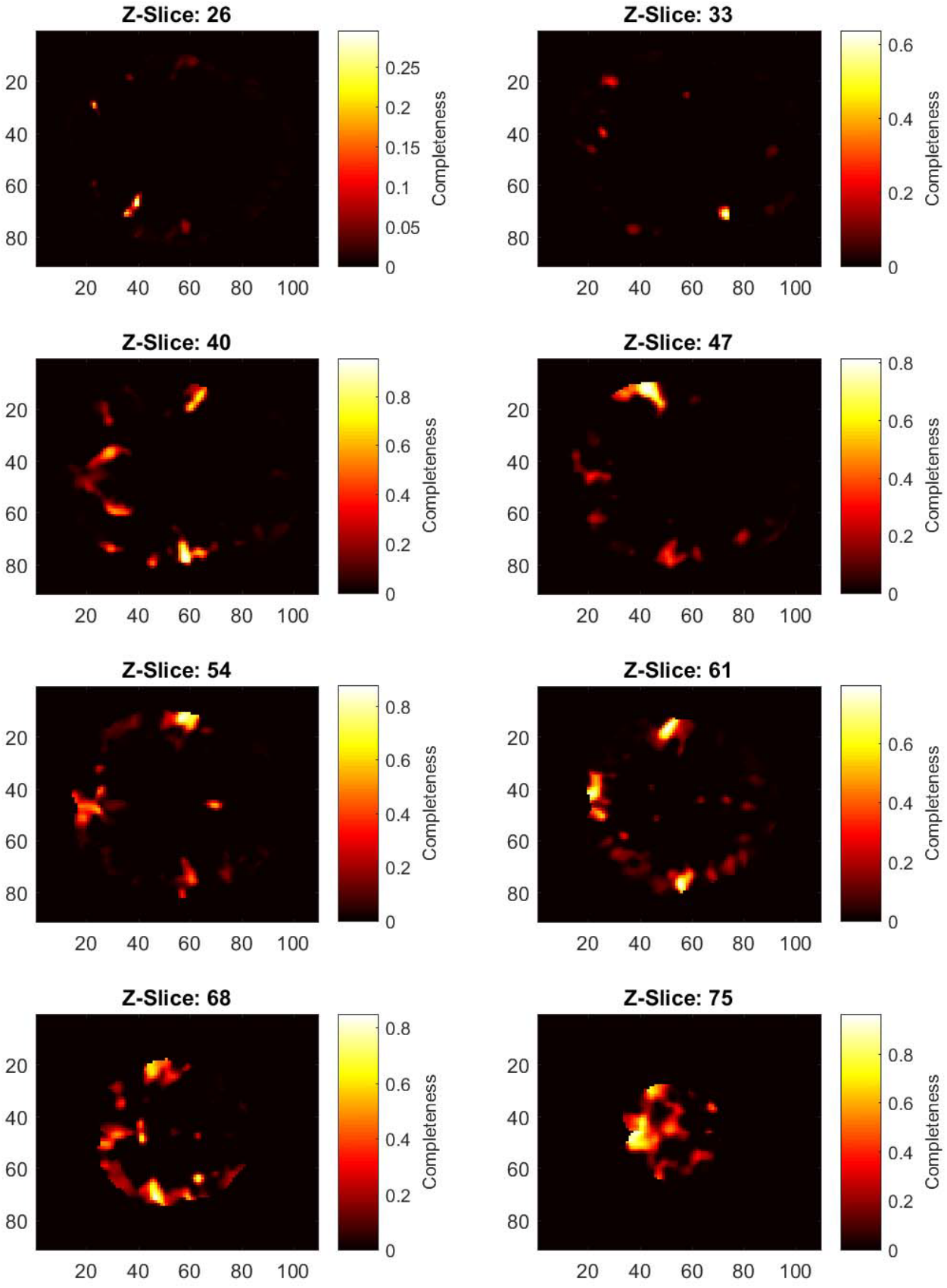


Figure 5.5: Connectedness of HCP Subject 102816 with r=0.63

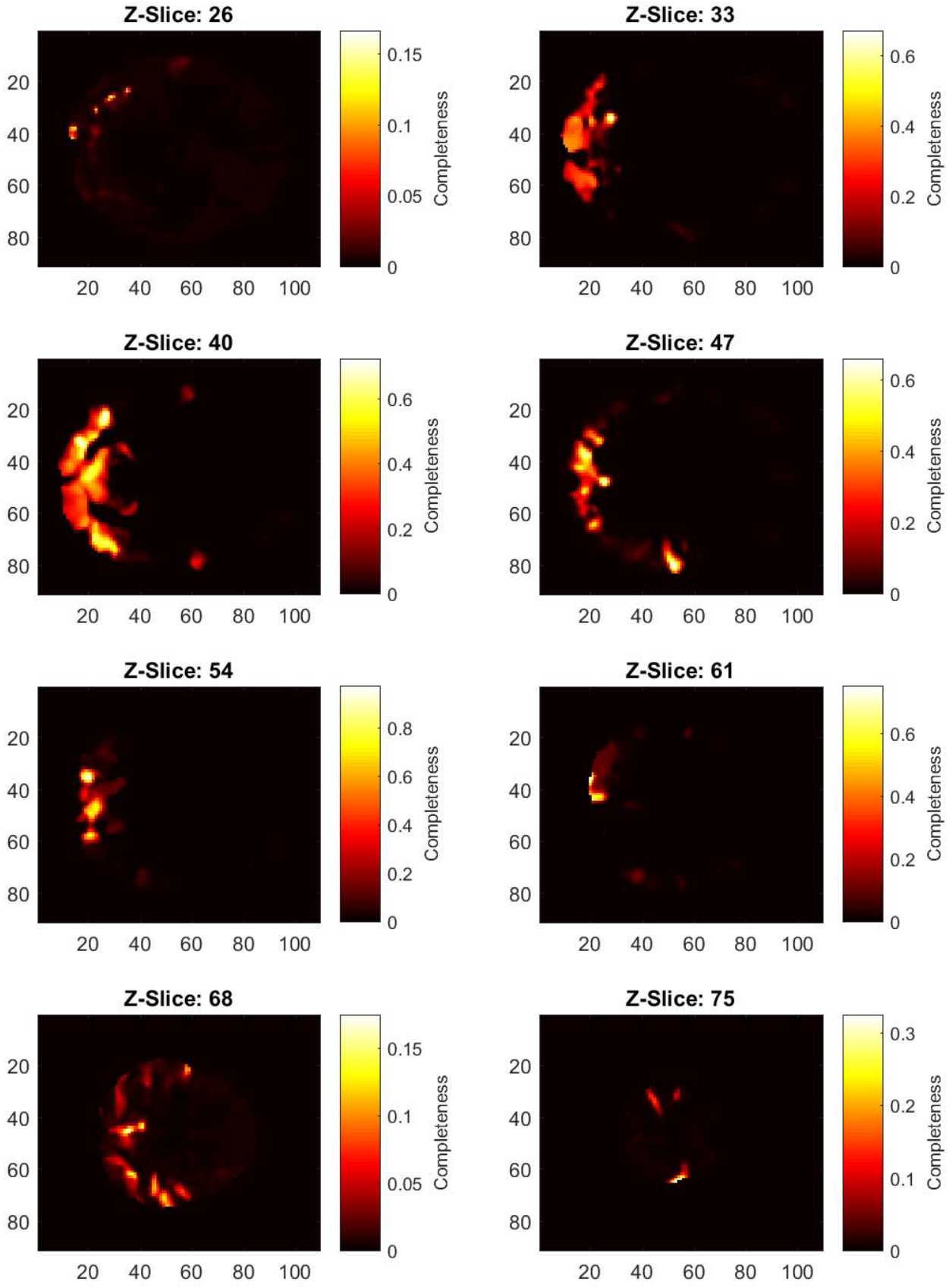


Figure 5.6: Connectedness of HCP Subject 112516 with r=0.70

Test #:	HCP Subject #:	# Voxels:	# Samples:	Load Time:	Process Time:	Save Time:	Total Time:	
1	102816	902629	1200	0.844	5.632	0.051	6.527	hrs
2	103818	902629	1200	0.843	5.632	0.132	6.607	hrs
3	104416	902629	1200	0.912	5.632	0.075	6.619	hrs
4	105014	902629	1200	0.841	5.631	0.080	6.552	hrs
5	105115	902629	1200	0.910	5.631	0.069	6.610	hrs
6	105923	902629	1200	0.843	5.631	0.078	6.552	hrs
7	106016	902629	1200	0.912	5.632	0.121	6.665	hrs
8	107422	902629	1200	0.844	5.632	0.098	6.574	hrs
9	108525	902629	1200	0.910	5.631	0.071	6.612	hrs
10	109325	902629	1200	0.842	5.631	0.086	6.559	hrs
11	111009	902629	1200	0.911	5.631	0.083	6.625	hrs
12	111413	902629	1200	0.845	5.632	0.092	6.569	hrs
13	111716	902629	1200	0.913	5.632	0.098	6.643	hrs
14	112314	902629	1200	0.842	5.631	0.113	6.586	hrs
15	112516	902629	1200	0.910	5.630	0.099	6.639	hrs
16	113215	902629	1200	0.844	5.632	0.072	6.548	hrs
17	113922	902629	1200	0.910	5.631	0.106	6.647	hrs
18	114924	902629	1200	0.845	5.632	0.098	6.575	hrs
19	116726	902629	1200	0.894	5.631	0.074	6.599	hrs
20	117122	902629	1200	0.844	5.632	0.086	6.562	hrs
21	118124	902629	1200	0.893	5.631	0.121	6.645	hrs
22	118225	902629	1200	0.842	5.631	0.090	6.563	hrs
23	119126	902629	1200	0.893	5.631	0.084	6.608	hrs
24	119833	902629	1055	0.740	4.990	0.097	5.827	hrs
			MEAN:	**0.868**	**5.605**	**0.091**	**6.563**	hrs
			STDDEV:	**0.041**	**0.128**	**0.019**	**0.158**	hrs

Processing Time Breakdown		
Data Loading (from SD Card)	13.22	%
RVP Core Processing	85.40	%
r Vector Storage (to SD Card)	1.40	%

Table 5.2: Processing time for HCP subjects with r=0.63

Test #:	HCP Subject #:	# Voxels:	# Samples:	Load Time:	Process Time:	Save Time:	Total Time:	
1	102816	902629	1200	0.844	5.632	0.031	6.507	hrs
2	103818	902629	1200	0.843	5.632	0.035	6.510	hrs
3	104416	902629	1200	0.912	5.632	0.030	6.574	hrs
4	105014	902629	1200	0.841	5.631	0.013	6.485	hrs
5	105115	902629	1200	0.910	5.631	0.034	6.575	hrs
6	105923	902629	1200	0.843	5.631	0.024	6.498	hrs
7	106016	902629	1200	0.912	5.632	0.031	6.575	hrs
8	107422	902629	1200	0.844	5.632	0.039	6.515	hrs
9	108525	902629	1200	0.910	5.631	0.039	6.580	hrs
10	109325	902629	1200	0.842	5.631	0.160	6.633	hrs
11	111009	902629	1200	0.911	5.631	0.027	6.569	hrs
12	111413	902629	1200	0.845	5.632	0.016	6.493	hrs
13	111716	902629	1200	0.913	5.632	0.026	6.571	hrs
14	112314	902629	1200	0.842	5.631	0.018	6.491	hrs
15	112516	902629	1200	0.910	5.630	0.029	6.569	hrs
16	113215	902629	1200	0.844	5.632	0.027	6.503	hrs
17	113922	902629	1200	0.910	5.631	0.038	6.579	hrs
18	114924	902629	1200	0.845	5.632	0.023	6.500	hrs
19	116726	902629	1200	0.894	5.631	0.035	6.560	hrs
20	117122	902629	1200	0.844	5.632	0.023	6.499	hrs
21	118124	902629	1200	0.893	5.631	0.022	6.546	hrs
22	118225	902629	1200	0.842	5.631	0.017	6.490	hrs
23	119126	902629	1200	0.893	5.631	0.030	6.554	hrs
24	119833	902629	1055	0.740	4.990	0.022	5.752	hrs
			MEAN:	**0.868**	**5.605**	**0.033**	**6.505**	hrs
			STDDEV:	**0.041**	**0.128**	**0.027**	**0.162**	hrs

Processing Time Breakdown

Data Loading (from SD Card)	13.34	%
RVP Core Processing	86.15	%
r Vector Storage (to SD Card)	0.51	%

Table 5.3: Processing time for HCP subjects with r=0.70

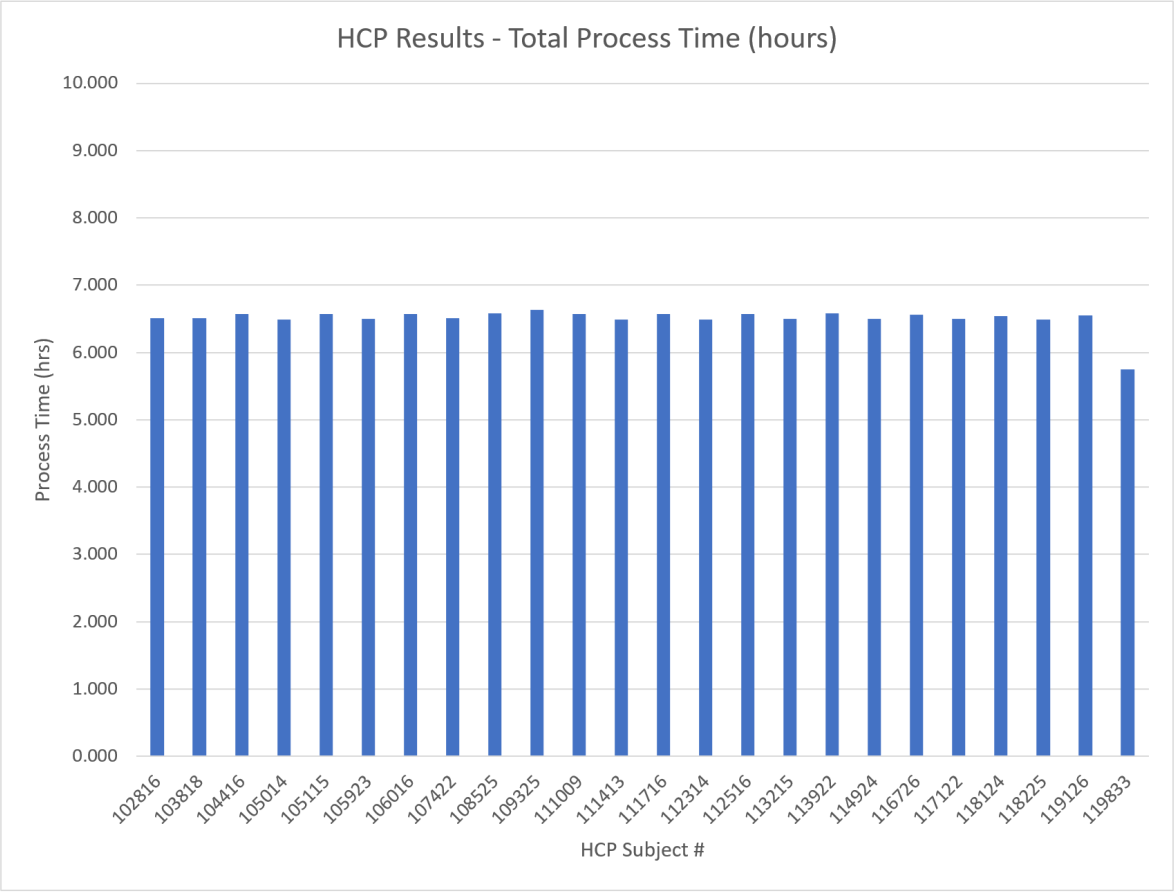


Figure 5.7: Graph of processing time for HCP subjects with r=0.63

5.3 PCC Results via CPU

A comparison of performance for the results generated by the HPCME is with a traditional CPU implementation. Numerous methods and algorithms can be used to optimize processing on a multi-core CPU and given that this research focused on developing an FPGA based solution, development of a high-performance multi-core algorithm was not developed. Instead, our comparisons are based on a two-fold approach. One of which is our C# implementation of the correlation computation, and the other based upon previously determined and published CPU results that have been extrapolated to compare to the dataset sizes in use in this research. Our analysis of published research focused on CPU performance results described in the GPU-PCC method [19], Minati's Fast Computation method [14], and Wang's Hybrid CPU-GPU Accelerated Framework process [40].

The implementation of the correlation algorithm was developed in Microsoft Visual Studio and written in C#. A test was done utilizing HCP subject #102816 with 902,629 voxels and 1,200 samples. The coding for the algorithm is shown in Figure 5.9 and implemented with single-precision variables (4-bytes). The algorithm did include the storage of the correlation results to a file as utilized only a single thread, which is a definitive factor is the performance of the algorithm. This algorithm was tested on a PC workstation with the following specifications:

- An NVIDIA GeForce GTX980 GPU
- Windows 10 OS
- Four-core Intel i7 CPU operating at 4.0 GHz
- 64 GB of DDR4 SDRAM

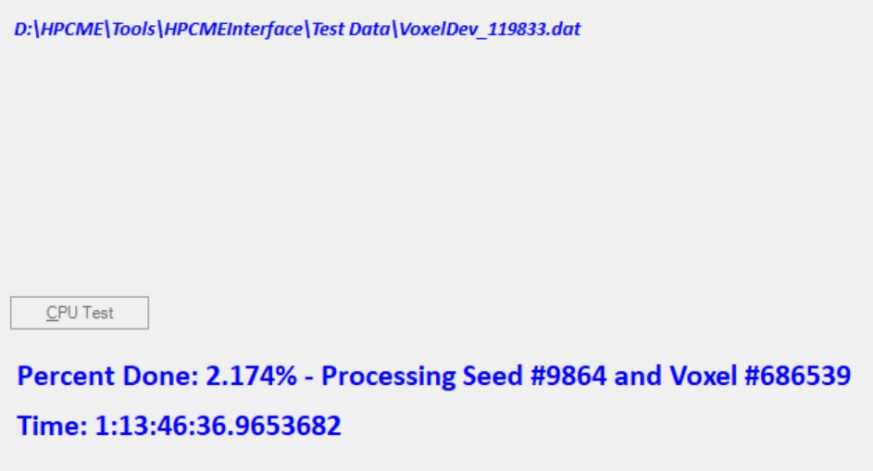


Figure 5.8: CPU Processing - 2.17% complete at 37 hours

As shown in Figure 5.8, the algorithm was run for 120 hours and achieved completion of the processing of 4.75%. Extrapolated, worst case, the CPU would complete this processing in approximately 98 days. Ideally, if an algorithm were developed using localized memory that was not shared between four threads, the expected performance would best case, be approximately 588 hours, or 24 days.

```
string fileResultName = pathData + "\\r-vector_CPU.bin";
if (File.Exists(fileResultName)) File.Delete(fileResultName);
FileStream fsRes = File.Open(fileResultName, FileMode.CreateNew);


//DateTime dtStart = DateTime.Now;
using (BinaryWriter bw = new BinaryWriter(fsRes))
{
    // Half (pair wise) and four systems / threads
    double max = ((double)iNumVoxels * (double)iNumVoxels) / 2.0;

    while (stopThreads == false)
    {

        for (int iSeedIndex = iSeedStart; iSeedIndex < iNumVoxels; iSeedIndex++)
        {
            int iAppCont = 0;
            iCurrentSeed = iSeedIndex;


            for (int iVoxelIndex = iSeedIndex; iVoxelIndex < iNumVoxels; iVoxelIndex++)
            {
                float fDeviationSum = 0.0F;
                iAppCont++;
                count++;
                if (iAppCont > 1000)
                {
                    percentDone1 = ((double)count / max) * 100.0;
                    Application.DoEvents();
                    iAppCont = 0;
                }

                for (int iSampleIndex = 0; iSampleIndex < iNumSamples; iSampleIndex++)
                {
                    fDeviationSum = fDeviationSum + (VoxelData[iSampleIndex, iVoxelIndex] * SeedData[iSampleIndex, iSeedIndex]);
                }

                // Get "r" value
                float r = fDeviationSum / (VoxelSDData[iVoxelIndex] * SeedSDData[iSeedIndex]);

                // Compare and save "r" vector if in excess
                if ((r > 0.7) & (r <= 1.0F))
                {
                    bw.Write(iSeedIndex);
                    bw.Write(iVoxelIndex);
                    bw.Write(r);
                }

                if (stopThreads == true) break;
            }

            if (stopThreads == true) break;

        }
    }
}
fsRes.Close();
```

Figure 5.9: CPU implementation of the correlation algorithm

Of the published CPU processing, there was a wide variance in results for the CPU performance with regards to the correlation of fMRI data. The number of voxels and time series length varied based upon the test as well as the CPU architecture and OS.

Minati's Fast Computation method [14] CPU comparison test used 227,808 voxels with a time series length of 284 samples from actual fMRI data from the HCP. This test ran two different test configurations. One configuration was with an Intel i7-3610QM 2.3 GHz processor (8 cores), running the Ubuntu Linux 12.04 OS in (32-bit). The other configuration was an Intel E5-2637 3.0 GHz processor (4 cores), running the CentOS Linux 6.5 (64-bit). With the 4-core configuration, the correlation was completed in 6,200 seconds, versus the 8-core configuration completing in 2,700 seconds. When extrapolated using the number of computations and the average computation time a 902,669 voxel by 1,200 sample test would be estimated to take 114 hours for the four-core and 50 hours for the 8-core. The extrapolation assumed a linear increase in the level of computation and did not take into consideration any additional loading, saving, or processing time to work with the higher resolution datasets.

The GPU-PCC method [19] CPU comparison test used a varying number of voxels, ranging from 10,000 to 100,000 voxels, with a time series length of 300 samples from a synthetic dataset. This test ran one test configuration, which consisted of two Intel Xeon E5 2620 2.4 GHz processors and 48 GB of RAM. With this computing configuration, the correlation with 40,000 voxels was completed in 725 seconds, for the 80,000 voxels in 2,850 seconds, and 100,000 voxels were completed in 4,500 seconds. When extrapolated using the number of computations and the average computation time, a 902,669 voxel by 1,200 sample test would be estimated to take approximately 405 hours.

For Wang's Hybrid CPU-GPU Accelerated Framework process [40] the CPU comparison test used 90,112 voxels and 58,523 voxels. With a time series length of 300 samples and 225 samples respectfully from an HCP fMRI dataset. This test ran two test configuration, which consisted of an Intel i7-3770 3.4 GHz quad-core CPU with 32 GB RAM. With this computing configuration, the correlation with 90,112 voxels, with a single thread, was completed in 1974 seconds, for the 58,523 voxels, with a single thread, was completed in 568 seconds, and for the 58,523 voxels, with eight threads, was completed in 130 seconds. When extrapolated using the number of computations and the average computation time, a 902,669 voxel by 1,200 sample test would be estimated to take

approximately 200 hours for a single thread and 46 hours for eight threads. All of the above results are shown in the Table 5.4.

Published CPU Test:	# Voxels:	# Samples:	Process Time:		# Voxels:	# Samples:	Estimated Time:	
4-core CPU (Minati, et. al.)	227808	284	6200	sec	902669	1200	114	hrs
8-core CPU (Minati, et. al.)	227808	284	2700	sec	902669	1200	50	hrs
2-core Xeon (GPU-PCC)	100000	300	4500	sec	902669	1200	407	hrs
2-core Xeon (GPU-PCC)	80000	300	2850	sec	902669	1200	403	hrs
2-core Xeon (GPU-PCC)	40000	300	725	sec	902669	1200	410	hrs
4-core i7 CPU 1-Thread (Wang, et. al.)	90112	300	1974	sec	902669	1200	220	hrs
4-core i7 CPU 1-Thread (Wang, et. al.)	58523	225	568	sec	902669	1200	200	hrs
4-core i7 CPU 8-Thread (Wang, et. al.)	58523	225	130	sec	902669	1200	46	hrs

Table 5.4: CPU processing time from published research. Extrapolation to HCP datasets used in this research

As previously mentioned the varying results of the various CPU tests, both performed by this research and in existing published research is significant. To compare the processed results of this research with the high resolution, high sample depth HCP datasets the extrapolation technique was derived from providing a best-case scenario, versus a more realistic case where other performance factors would slow the CPU processing, such as storage and other processing threads and background processes. In the above extrapolation, the best case scenario was using an Intel i7-3770 3.4 GHz quad-core CPU with 32 GB RAM running 8-threads and processing the HCP dataset in 46 hours, or approximately two days. The average processing time across all methods both published and completed by this research provided a correlation processing in 271 hours or approximately eleven days.

5.4 Results of GPU-PCC Method

The GPU-PCC method, which is a CUDA based GPU algorithm for computing the Pearson's Correlation Coefficient (PCC) is described in Section 1.5.

The developers of the GPU-PCC methods, Eslami et al., presented their research with the performance evaluation as summarized below[19].

They ran their experiments on a Linux server with the following performance specifications:

- An NVIDIA Tesla K40c GPU
- Ubuntu Server OS 14.01
- Two-core Intel Xeon E5 2620 operating at 2.4 GHz
- 48 GB of SDRAM

The NVIDIA Tesla K40c has fifteen streaming multiprocessors each with 192 CUDA cores, for a total of 2,880 CUDA cores, and 11.25 GB of global memory.

Eslami et al. compared their algorithms performance to three other methods. The first being matrix-vector multiplication, the second comparing the results to a CPU implementation of PCC, and the third being a hybrid GPU-GPU method by Wang et al. [40]. The CPU results for these tests are described in Section 5.3. They performed experiments using both synthetic and actual fMRI datasets.

		Time (sec)											
Number of Voxels (N)	Sample Size (M)	Run #1	Run #2	Run #3	Run #4	Run #5	Run #6	Run #7	Run #8	Run #9	Run #10	Average	SD
20000	300	2.70	1.72	2.70	2.69	2.71	1.70	1.68	1.69	2.69	2.71	2.30	0.49
30000	300	3.75	4.74	3.76	4.76	3.91	4.73	4.74	4.73	3.73	3.74	4.26	0.48
40000	300	6.63	7.60	6.64	7.63	6.62	6.64	6.66	6.60	6.61	6.60	6.82	0.40
50000	300	12.37	11.26	11.25	11.25	11.25	11.30	11.28	11.28	12.27	11.23	11.47	0.42
75000	300	23.70	23.68	22.72	23.69	23.69	23.66	24.69	24.69	25.66	25.67	24.18	0.91
90000	300	33.57	34.54	33.52	32.52	32.47	35.47	35.48	35.55	35.59	36.46	34.52	1.34
100000	300	40.99	41.01	41.98	40.98	42.06	42.06	45.02	44.00	45.01	43.99	42.71	1.55
125000	300	67.24	65.23	67.22	64.24	65.64	71.24	70.47	70.41	68.25	67.22	67.72	2.25
150000	300	92.98	91.31	92.50	91.45	97.30	95.29	98.40	96.33	94.33	96.31	94.62	2.37
175000	300	130.66	124.70	127.93	132.69	129.40	133.40	132.44	135.61	130.45	136.48	131.38	3.36
200000	300	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL		

Table 5.5: GPU-PCC performance with synthetic fMRI data (GTX980, M=300)

Their synthetic datasets were created with voxels sizes of 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000 and with the time-course sample size of 300. For each vector, that generated a random floating point numbers in the range of -2 to 2. For the real fMRI dataset, they used the Orangeburg dataset [41].

This research comparison utilizes the published source code for GPU-PCC on a PC workstation and has the following specifications:

- An NVIDIA GeForce GTX980 GPU
- Windows 10 OS
- Four-core Intel i7 CPU operating at 4.0 GHz
- 64 GB of DDR4 SDRAM

The NVIDIA GTX980 has 2,048 CUDA cores, operating at 1,126 MHz, and has 4 GB of global memory (GDDR5) with a memory bandwidth of 224 GB/sec. It should be noted that this GPU configuration is not as capable as the solution that Eslami et al. utilized in their experiments.

Also, given the configuration of the available source code, it was only possible to run a test using synthetic data as the source code that we ran in our experiments. It did not allow for the importing of real fMRI datasets. We ran a similar set of ten tests where we changed the voxel size of the data and kept the sample depth fixed. We used synthetic datasets with voxels sizes of 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000, 125000, 150000, 175000, and 200000 and with the time-course sample sizes of 300 and 1200. The results of the tests with a time-course sample depth of M = 300 is shown below in Table 5.5, and the results of the tests with a time-course sample depth of M = 1200 is shown below in Table 5.6.

As is illustrated in Table 5.5, the GPU-PCC algorithm performed well with a sample size of M=300 for voxels sizes up to 175,000 voxels. At 200,000 voxels the algorithm would not operate correctly and crashed. Hence it did not return any results. A plot of the average process time of the ten independent runs versus the voxel size is shown in Figure 5.10.

		Time (sec)											
Number of Voxels (N)	Sample Size (M)	Run #1	Run #2	Run #3	Run #4	Run #5	Run #6	Run #7	Run #8	Run #9	Run #10	Average	SD
20000	1200	4.27	4.31	5.29	5.26	4.27	4.26	4.27	4.27	5.27	5.27	4.67	0.49
30000	1200	9.43	10.41	9.40	FAIL	9.44	9.42	9.42	9.40	10.41	FAIL	9.67	0.43
40000	1200	17.63	FAIL	FAIL	FAIL	FAIL	FAIL	17.61	17.63	FAIL	FAIL	17.62	0.01
50000	1200	FAIL	26.76	26.75	26.78	26.81	FAIL	27.79	FAIL	27.79	FAIL	27.11	0.48
75000	1200	FAIL	FAIL	60.35	60.29	60.29	60.31	58.77	FAIL	FAIL	FAIL	60.00	0.62
90000	1200	FAIL	FAIL	86.24	86.20	86.14	FAIL	FAIL	FAIL	FAIL	FAIL	86.19	0.04
100000	1200	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL		
125000	1200	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL		
150000	1200	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL		
175000	1200	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL		
200000	1200	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL		

Table 5.6: GPU-PCC performance with synthetic fMRI data (GTX980, M=1200)

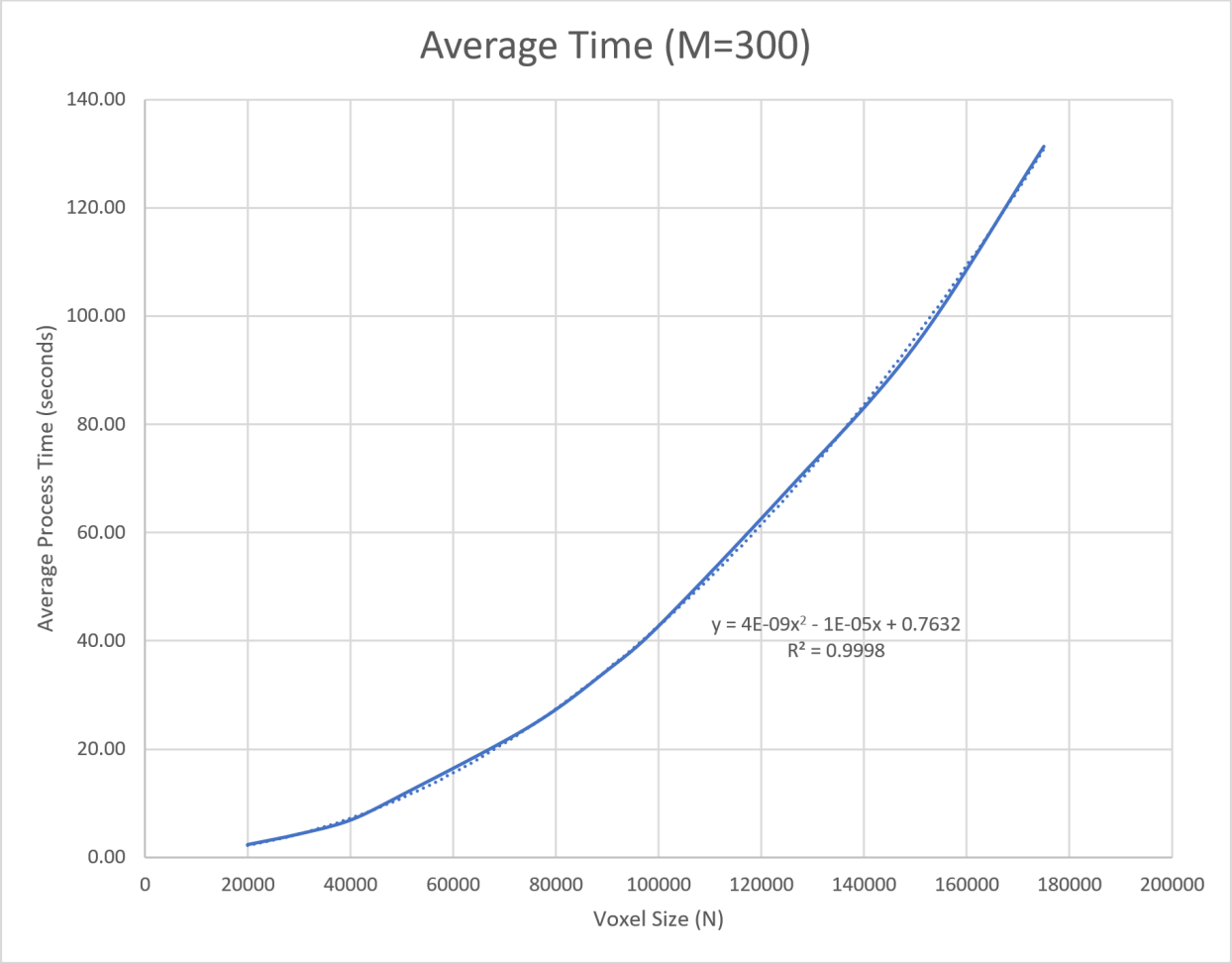


Figure 5.10: GPU-PCC performance with synthetic fMRI data (GTX980, M=300)

The results that we generated compared similarly to the results that were compiled by Eslami et al., actually within a few seconds of deviation. For larger voxels sizes the standard deviation of

the ten runs were approximately 4% of the process time. It is also of note that these results are just for the processing of the correlation matrix, and not any other facet of the computation of the final result set, including saving the results to a non-volatile file for post-processing.

A regression of the process time was done to help predict what the processing time would be for a given voxel size with a fixed sample depth, in this case, M=300. The regression equation was determined to be the following, with N being the number of voxels:

$$Time(N)_{sec} = 4 \times 10^{-9} N^2 - 1 \times 10^{-5} N + 0.7632 \tag{5.1}$$

Given this above regression, it was determined that if using the extrapolated results and given a GPU application and architecture that could support a successful operation at higher voxel lengths, the expected performance of the GPU-PCC method would be able to compute the correlation matrix in approximately 3,251 seconds or 0.903 hours.

As previously shown in Table 5.6 the GPU-PCC algorithm did not perform well with a sample size of M=1,200 for voxels sizes over to 20,000 voxels. At 30,000 voxels and above, the algorithm would not operate correctly and routinely would crash the workstation. Hence it did not return any results. A plot of the average process time of the successful runs versus the voxel size is shown in Figure 5.11.

Since the success of the algorithm to produce results above 90,000 voxels, it was determined that the extrapolation to 902,629 voxels would be unreliable and a regression not warranted.

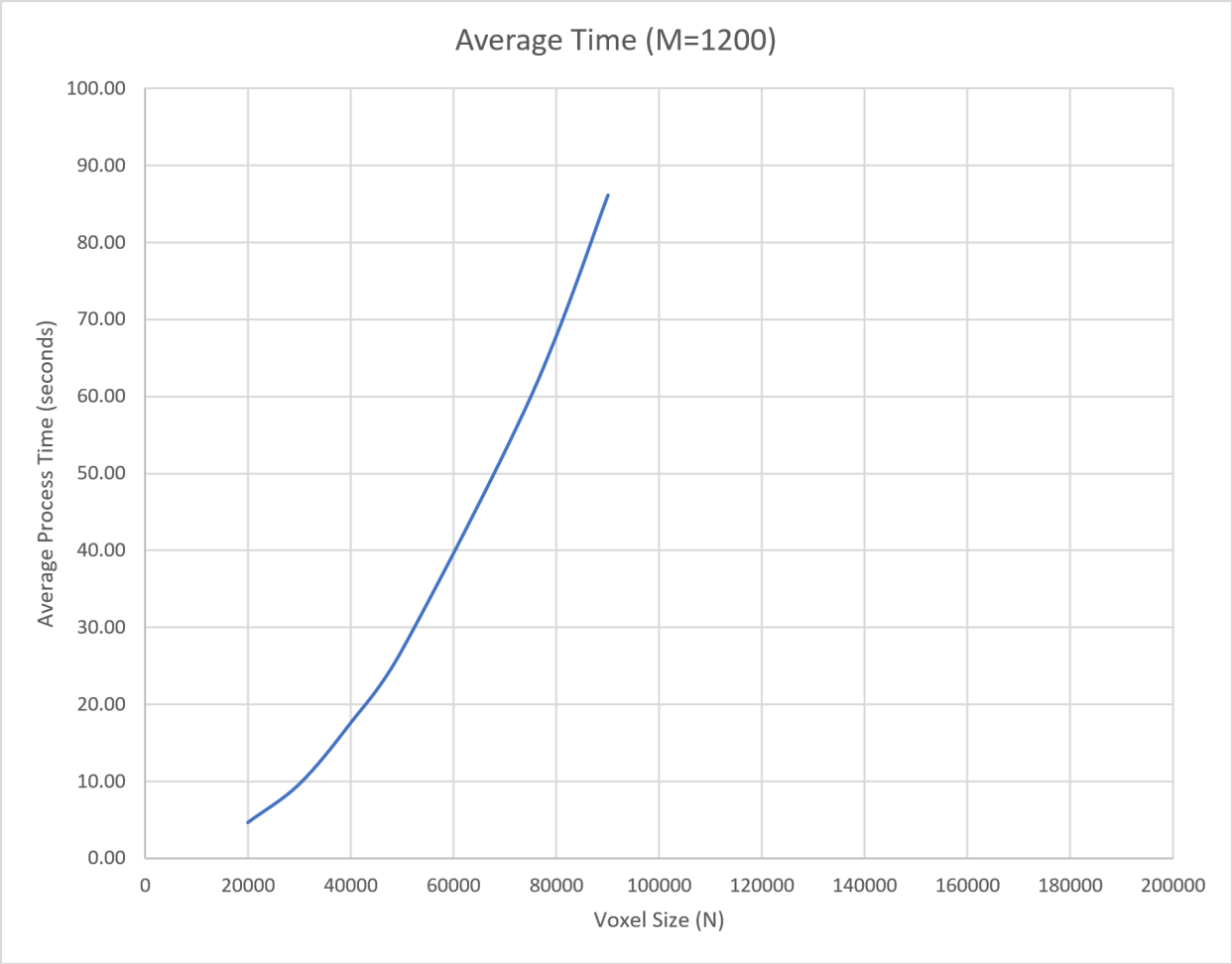


Figure 5.11: GPU-PCC performance with synthetic fMRI data (GTX980, M=1200)

5.5 Results of CONN Toolbox

The HCP datasets provided by Dr. Orr underwent additional processing and analysis using the CONN toolbox [28]. CONN is a Matlab-based application designed for functional connectivity analysis. CONN was compiled as a standalone application for MATLAB R2016b in CentOS six running on a 128-core Intel Xeon Broadwell blade cluster.

Dr. Orr's configuration for preprocessing in CONN consisted of structural segmentation, smoothing (6mm FWHM), and artifact detection (global signal z-value threshold: 5, framewise motion threshold: 0.9 mm). Data was then denoised with linear regression with confound regressors for five temporal components each from the segmented CSF and white matter, 24 motion realignment parameters, signal and motion outliers, and the 1st order derivative from the effect of rest. Finally, data underwent linear detrending and bandpass filtering (0.01 - 0.1 Hz).

Datasets were initially loaded and the timeseries extracted for each ROI. The ROIs were determined from using the Harvard-Oxford Atlas [42], which has 132 ROIs. Data is then checked for consistency and then denoised. These datasets were then processed using the using first a ROI-to-ROI analysis, then a ROI-to-voxel.

The CONN toolkit performed the Voxel-to-Voxel global correlation analysis using the full group of 24 subjects. A Voxel-to-Voxel global correlation analysis was performed as this was the most comparable analysis to the HPCME system [43]. The Voxel-to-Voxel analysis is computed using SVD, getting the 64 most significant eigenvectors and eigenvalues. This is done as it is concluded that it is impractical to store the full correlation matrix [44]. The processing time on a iMac, with a Intel i5 CPU and 16 GB of 1600 MHz DDR3 memory, to perform these steps are shown in Table 5.7.

Processing Step:	Duration:	
Preprocesing	28	min
Setup	127	min
Denoising	4	min
First-Level ROI	5	min
First-Level V2V GC	11	min
TOTAL:	175	min
	2.92	hours

Table 5.7: CONN Processing Time

An overall comparison was performed to the averaged results from the HPCME. This comparison was to being to understand what general agreement there was between these two different methods and to understand if the HPCME would generate results that drastically departed with results generated by CONN.

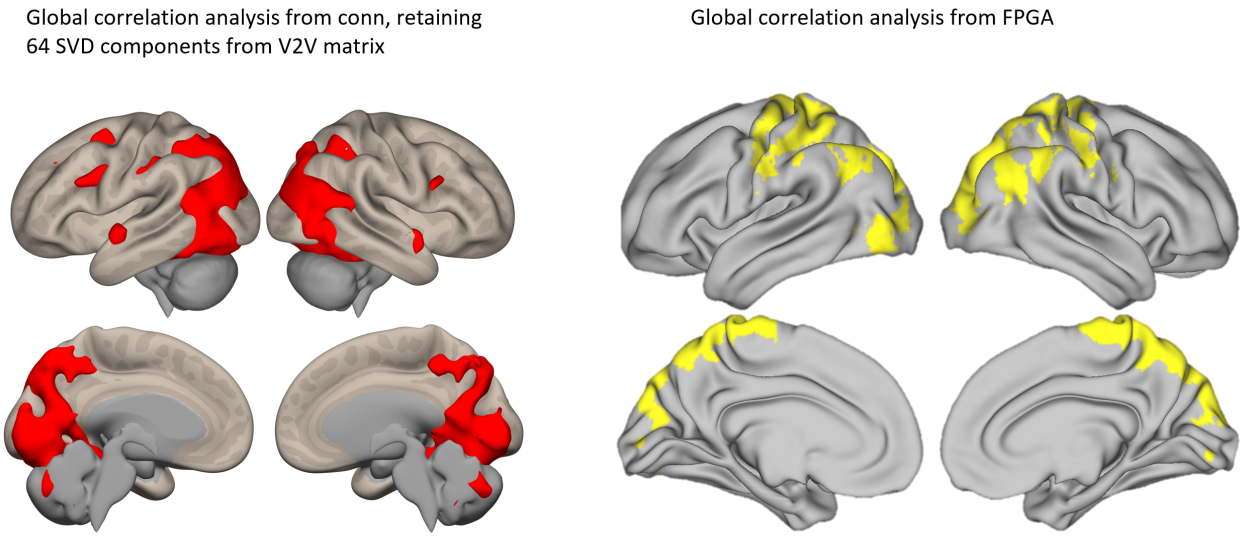


Figure 5.12: CONN to HPCME comparison - side-by-side

Figure 5.12 shows a comparison of the correlation results between the CONN processing and the HPCME. The CONN results, shown in red, are the 64 SVD components from a Voxel-to-Voxel

correlation. For the HPCME (FPGA), the average of 24 subjects connectedness correlation results are shown in yellow. The top left image shows the left lateralization, the bottom left is the left medial, the top right is the right lateralization, and the bottom right is the right medial. The purpose of this comparison was to being to understand where there was alignment and disagreement between the two methods. It should be noted that for this test the HPCME (FPGA) used non-prepossessed and non-denoised files.

An overlay of the CONN toolbox correlation results and the HPCME (FPGA) results are shown in 5.13. These results shown where the HPCME found correlations that CONN did you show, and CONN found correlations that the HPCME did not. It should be noted that there is good agreement on the regions that overlap. These results, as will be further discussed in the following section, show that future research will be needed to explore this fully not that a method has been developed to produce rapid correlation of the full resolution HCP datasets.

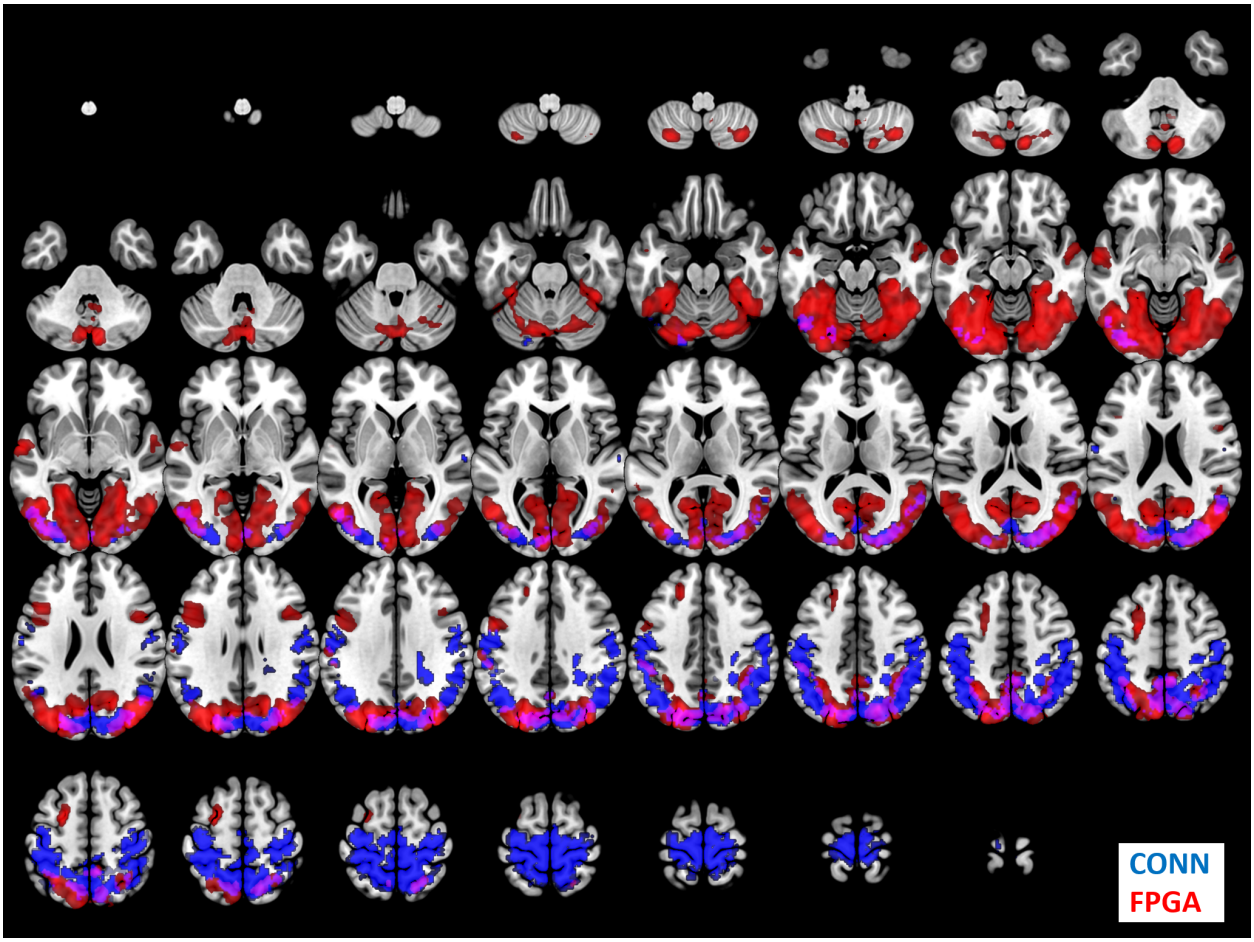


Figure 5.13: CONN to HPCME comparison - overlapping

Theses results showed overlap and also divergence. To access the similarity Dice's Similarity Coefficient (DSC) was computed between the resulting average connectedness maps, as defined by the following equation for a binary set of A and set B:

$$DSC = \frac{2|A \cap B|}{|A| + |B|} \tag{5.2}$$

The chart in Figure 5.8 shows the Dice coefficient for various functional areas of the brain. The results of this are further discussed in the following section.

Network	Dice Coefficient
Default Mode	0.161
Sensori-Motor	0.307
Visual	0.222
Salience	0.004
Dorsal Attention	0.143
Fronto-Parietal	0.050
Language	0.013
Cerebellar	0.003

Table 5.8: CONN to HPCME comparison - Dice's coefficient

6. DISCUSSION

This research aimed at developing a new processing architecture that would accelerate the processing of high-resolution brain connectivity maps from HCP datasets. In this section, the questions of the comparison of the results for the various results previous shown will be discussed. This section will also examine possible future research based on these results.

6.1 HPCME

The HPCME was utilized to process twenty-five high-resolution, high sample depth connectedness maps with data for the HCP. This processing took less than ten days, and with a mean processing time of 6.5 hours.

The breakdown of the processing time, shown in Figure 6.1, shows that the data loading and r-vector storage overhead only account for 13.2% and 1.4% of the total time respectively. This low overhead rate shows that as an architecture the HPCME is efficient in processing. Future development can improve this overhead by using NVMe FLASH for results and a 256-bit wide, high-speed 8GB of DDR4 SDRAM configuration for dataset storage.

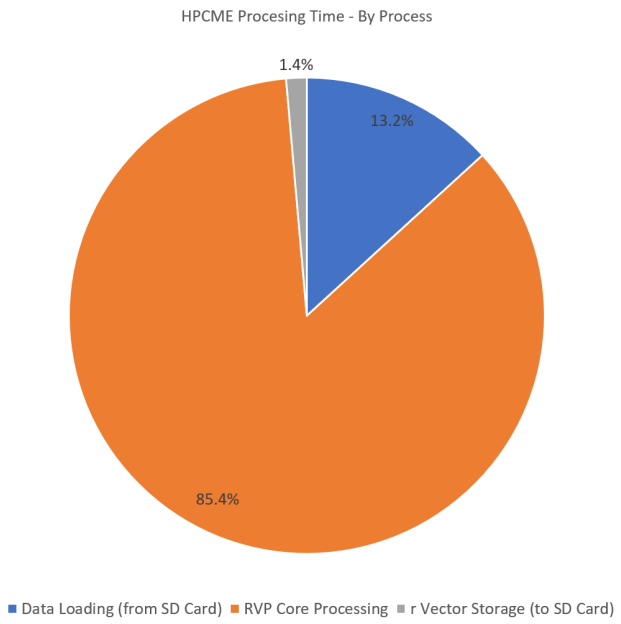


Figure 6.1: HPCME data processing - breakdown by task (overhead)

An important characteristic of the HPCME that is shown in the results is the processing time was consistent and had a standard deviation of less than 2.5%, even with changing correlation threshold coefficients. This consistent processing is due primarily to the low overhead rate and the time to save r-vectors to FLASH storage.

The HPCME, having near fully independent NDEs, enables the performance of the overall system to scale-up and is only limited, within reason, to the number of NDEs that are physically practical. Given that each NDE consumes about 35 watts and requires only minimal operational support it would be possible to include eight or sixteen NDEs into one HPCME assembly. This configuration would thereby reduce the processing time of these same HCP datasets from 6.5 hours to approximately 3.25 hours and 1.625 hours respectively.

Given the obtained results, the HPCME can be used in a lower-resolution or region based processing application. With a region based processing the processing time can be lowered to a point where it would be possible to perform near "real time" processing, with processing measured in seconds. This configuration would enable the HPCME to be used in real time neurological feedback applications and DFC analyses.

The HPCME is currently designed to performed seed-based correlation analysis (VVCA). However, other algorithms can be implemented in future work by modification of the mathematical ALU pipeline. For instance, a future application would be to use seed-based analysis with diffusion-weighted imaging (DWI) for tractography instead of the currently implemented fMRI and rs-fMRI analyses. In fact, it would be possible to use a script to reconfigure the HPCME to load the specific BIT file corresponding to the selected analysis method to be performed.

The computational performance of the HPCME is based upon the number of operations that are being performed at any one point. Examining the high-resolution, high sample rate HCP data sets with a voxel size of 902,629 voxels and 1,200 samples each. The overall performance of the HPCME platform is a sustained data stream processing rate of 2.89 TFLOPs and projected to be 11.6 TFLOPS with a sixteen core HPCME. It is important to stress that this processing rate is based on a continually streamed random set of data.

6.2 Comparison to GPU Methods

The GPU-PCC method, when compared to the HPCME has some advantages. For instance, the GPU-PCC method, relying solely on existing GPUs enables the development of a high-performance computing system without customized hardware and just a software (CUDA) implementation.

As shown in the results from section 5.4 the performance of the GPU-PCC method with a small voxel size is impressive, and in comparison to the HPCME, even superior, given a superior GPU such as a Nvidia Kepler. However, as was noted previously, with the high-resolution datasets the GPU, used for our tests, resulted in an unstable workstation and failures. Also, the GPU-PCC algorithm, as provided, did not provide an adequate method to copy the final results to a storage medium. This factor, as previously discussed, is a critical factor as the correlation matrix for 90,6929 voxels is on the order of 4 TB.

The use of GPUs to accelerate the generation of high-resolution brain connectivity maps is still a very active area of research and should not be disregarded in favor of FPGAs. Future research may be focused on combining the benefits of the multitude of GPU cores with the customized and pipelined architecture of an FPGA. Thus, enabling the development of a hybrid solution of a

standalone HCP processing architecture.

6.3 CPU Comparison

CPUs offered the most divergence results of the methods, compared to the processing of HCP datasets to produce brain connectivity maps, shown in section 5.3. As shown previously, a CPU algorithm can perform correlation at the fasts of 46 hours for an eight thread implementation to upwards of 515 hours for a four-core implementation done in this research study. We can only speculate as to what can lead to such a large divergence, but one theory is due to the memory organization of test data for correlation. To help understand this divergence, especially with the outliers that a significantly lower than expected processing time, such as the before mentioned 46-hour result, it is plausible that the data used was not actual HCP results, but a synthetic test array. This data could then be stored in the local cache of the microprocessor, thus enabling a CPU to have much higher performance in processing. It is also plausible that the same memory regions were being accessed from global memory, again giving the CPU an advantage in processing.

The CPU processing method tested in this research study used the same HCP dataset that was used in the HPCME tests, thereby ensuring a consistent comparison. The CPU processing algorithm that was developed is not considered the most efficient, however it is focused strictly on the processing of the correlation coefficient.

Other methods showed similar performance to what was determined by this research. For instance, as was shown in the CPU results, the performance determined by the researchers for the GPU-PCC method showed extensive processing times for multi-core processing using Intel Xeon processors. Given that this research did use both HCP and synthetic datasets this may lead credence to their CPU algorithm being closer to an actual CPU performance metric. To put in comparison the GPU-PCC research claims that the GPU-PCC algorithm is 94.62 times faster than the traditional CPU methods and 4.28 times faster than the existing GPU based techniques on a fMRI dataset with up to 90,000 voxels [19].

To have as fair a comparison as possible, given this divergence of CPU results, an average of the various methods was used at the extrapolated dataset size that the HPCME was processing,

which resulted in an average processing time of 271 hours.

The primary limitation of CPU processing is not in the ALU capability, but instead in the memory architecture. The correlation algorithm, as discussed in section 1.3, is not complex, but instead requires a near constant stream of voxel and seed data to be efficient due to the large size of the correlation matrix. This processing architecture is not ideal when CPU memory access is random and not burst mode.

The HPCME system, when compared to the various CPU implementations, is shown in Table 6.1. On average the HPCME is 40 times faster in processing, and seven times faster when compared to the fastest implementation, regardless of the divergence noted. The HPCME demonstrates a clear and consistent advantage over using CPU based processing methods.

CPU Test:	# Voxels:	# Samples:	Estimated Time:		HPCME Speedup
4-core i7 CPU 4-Thread	902669	1200	515	hrs	79
4-core CPU (Minati, et. al.)	902669	1200	114	hrs	18
8-core CPU (Minati, et. al.)	902669	1200	50	hrs	8
2-core Xeon (GPU-PCC)	902669	1200	407	hrs	63
2-core Xeon (GPU-PCC)	902669	1200	403	hrs	62
2-core Xeon (GPU-PCC)	902669	1200	410	hrs	63
4-core i7 CPU 1-Thread (Wang, et. al.)	902669	1200	220	hrs	34
4-core i7 CPU 1-Thread (Wang, et. al.)	902669	1200	200	hrs	31
4-core i7 CPU 8-Thread (Wang, et. al.)	902669	1200	46	hrs	7
				MEAN	**40**

Table 6.1: HPCME vs. CPU Methods

6.4 CONN Toolkit

The CONN toolbox was used to provide a general comparison to the results generated by the HPCME. Given the multitude of analysis methods available through the CONN toolbox, a detailed and exacting comparison was not warranted for initial comparisons. However, future research will focus on detailed comparisons and in determining the nature of diverging results between the CONN toolbox's connectedness maps and those generated by the HPCME.

For instance, when comparing the Dice's coefficient of the various regions, shown in Table 6.2,

there was a low similarity. However, there was more similarity in sensorimotor and visual networks. A plausible reason for this lower than expected similarity was because the HPCME results were generated with unprocessed datasets, whereas the CONN toolbox did perform extensive preprocessing of the data before the correlation processing, as discussed previously. Also, it was noted that there was significant variation in the connectedness maps between the various subjects. Since the CONN toolbox and HPCME results were combined for this analysis, this likely influenced these results looking for similarity.

Network	Dice Coefficient
Default Mode	0.161
Sensori-Motor	0.307
Visual	0.222
Salience	0.004
Dorsal Attention	0.143
Fronto-Parietal	0.050
Language	0.013
Cerebellar	0.003

Table 6.2: CONN to HPCME comparison - Dice's coefficient

The HPCME results do broadly agree with those from the CONN SVD method in that the HPCME did not generate connectedness in functional areas that CONN did not. Future research will include examining the similarity of individual comparison of HCP datasets and not the combined set. Also, future research will examine other comparisons to other analytical methods that the CONN toolkit provides, such as seed-based correlations, ROI-to-ROI graph analyses, and others.

7. CONCLUSION

This research aimed to design, produce, and test a High-Performance Correlation and Mapping Engine (HPCME) that enabled high throughput processing of brain connectivity data sets. The HPCME system is a computer workstation and a high-performance FPGA co-processing engine optimized for computing high volume correlation data.

By reducing the processing time to generate brain connectivity maps, neuroscience researchers will be able to understand the various statistical links, thereby enabling them to be able to determine the effect of aging, Alzheimer's, addiction, schizophrenia, dyslexia, autism, and ADHD. Also, being able to perform near real-time processing of fMRI data sets will enable DFC research to be performed more rapidly over a more significant segment number of nodes than currently [5].

This research achieved the following objectives:

1. Design and build a proof-of-concept HPCME hardware platform.

2. Expand the capability of the HPCME system to achieve the targeted performance objectives with multiple FPGAs.

3. Validate and demonstrate the HPCME with various datasets from the HCP and compare the results generated by using existing connectivity toolkits.

The HPCME system, as designed, adequately demonstrates the ability to generate voxel-to-voxel brain network connectivity maps within seven hours or less from the high-resolution, high sample size, HCP datasets. These test results show that HPCME with four FPGAs could improve the correlation processing speed by a factor of 40 or more over that of a PC workstation with a multicore CPU. This architecture and method can be used as a powerful tool to rapidly process the brain connectome data at finer resolutions, which potentially could lead to discoveries in diagnosis and treatment of neurological diseases and cognitive degradation.

REFERENCES

[1] F. X. Castellanos, A. D. Martino, R. C. Craddock, A. D. Mehta, and M. P. Milham, "Clinical applications of the functional connectome," *NeuroImage*, vol. 80, no. Supplement C, pp. 527 – 540, 2013. Mapping the Connectome.

[2] J. Elam, "Lifespan pilot report available, foas for lifespan development, aging announced," *Human Connectome Project*, 2015.

[3] D. C. V. Essen, S. M. Smith, D. M. Barch, T. E. Behrens, E. Yacoub, and K. Ugurbil, "The wu-minn human connectome project: An overview," *NeuroImage*, vol. 80, no. Supplement C, pp. 62 – 79, 2013. Mapping the Connectome.

[4] D. V. Essen and K. Ugurbil, "The future of the human connectome," *NeuroImage*, vol. 62, no. 2, pp. 1299 – 1310, 2012. 20 YEARS OF fMRI.

[5] D. Akgun, U. Sakoglu, J. Esquivel, B. Adinoff, and M. Mete, "Gpu accelerated dynamic functional connectivity analysis for functional mri data," *Computerized Medical Imaging and Graphics*, vol. 43, no. Supplement C, pp. 53 – 63, 2015.

[6] X.-N. Zuo, Y. He, R. F. Betzel, S. Colcombe, O. Sporns, and M. P. Milham, "Human connectomics across the life span," *Trends in Cognitive Sciences*, vol. 21, no. 1, pp. 32 – 45, 2017.

[7] F. Krause, C. Benjamins, M. LÃijhrs, J. Eck, Q. Noirhomme, M. Rosenke, S. Brunheim, B. Sorger, and R. Goebel, "Real-time fmri-based self-regulation of brain activation across different visual feedback presentations," *Brain-Computer Interfaces*, vol. 4, no. 1-2, pp. 87–101, 2017.

[8] M. S. Spetter, R. Malekshahi, N. Birbaumer, M. LÃijhrs, A. H. van der Veer, K. Scheffler, S. Spuckti, H. Preissl, R. Veit, and M. Hallschmid, "Volitional regulation of brain responses

to food stimuli in overweight and obese subjects: A real-time fmri feedback study," *Appetite*, vol. 112, no. Supplement C, pp. 188 – 195, 2017.

[9] J. Richiardi, H. Eryilmaz, S. Schwartz, P. Vuilleumier, and D. V. D. Ville, "Decoding brain states from fmri connectivity graphs," *NeuroImage*, vol. 56, no. 2, pp. 616 – 626, 2011. Multivariate Decoding and Brain Reading.

[10] C. R. Cameron, J. Saad, Y. Chao-Gan, V. J. T, C. F. Xavier, and D. M. Adriana, "Imaging human connectomes at the macroscale," *Nature Methods*, vol. 10, no. 524, 2013.

[11] S. M. Smith, D. Vidaurre, C. F. Beckmann, M. F. Glasser, M. Jenkinson, K. L. Miller, T. E. Nichols, E. C. Robinson, G. Salimi-Khorshidi, M. W. Woolrich, D. M. Barch, K. UÄ§urbil, and D. C. V. Essen, "Functional connectomics from resting-state fmri," *Trends in Cognitive Sciences*, vol. 17, no. 12, pp. 666 – 682, 2013. Special Issue: The Connectome.

[12] M. P. van den Heuvel and H. E. H. Pol, "Exploring the brain network: A review on resting-state fmri functional connectivity," *European Neuropsychopharmacology*, vol. 20, no. 8, pp. 519 – 534, 2010.

[13] P. J. Olesen, Z. Nagy, H. Westerberg, and T. Klingberg, "Combined analysis of dti and fmri data reveals a joint maturation of white and grey matter in a fronto-parietal network," *Cognitive Brain Research*, vol. 18, no. 1, pp. 48 – 57, 2003.

[14] L. Minati, D. ZacÃă, L. DâĂŹIncerti, and J. Jovicich, "Fast computation of voxel-level brain connectivity maps from resting-state functional mri using l1-norm as approximation of pearson's temporal correlation: Proof-of-concept and example vector hardware implementation," *Medical Engineering and Physics*, vol. 36, no. 9, pp. 1212 – 1217, 2014.

[15] R. N. Boubela, K. Kalcher, W. Huf, C. Nasel, and E. Moser, "Big data approaches for the analysis of large-scale fmri data using apache spark and gpu processing: A demonstration on resting-state fmri data from the human connectome project," *Frontiers in Neuroscience*, vol. 9, p. 492, 2016.

[16] K. Loewe, M. Grueschow, C. M. Stoppel, R. Kruse, and C. Borgelt, "Fast construction of voxel-level functional connectivity graphs," *BMC Neuroscience*, vol. 15, p. 78, Jun 2014.

[17] L. Minati, M. Cercignani, and D. Chan, "Rapid geodesic mapping of brain functional connectivity: Implementation of a dedicated co-processor in a field-programmable gate array (fpga) and application to resting state functional mri," *Medical Engineering and Physics*, vol. 35, no. 10, pp. 1532 – 1539, 2013.

[18] R. Martuzzi, R. Ramani, M. Qiu, X. Shen, X. Papademetris, and R. T. Constable, "A whole-brain voxel based measure of intrinsic connectivity contrast reveals local changes in tissue connectivity with anesthetic without a priori assumptions on thresholds or regions of interest," *NeuroImage*, vol. 58, no. 4, pp. 1044 – 1050, 2011.

[19] T. Eslami, M. G. Awan, and F. Saeed, "Gpu-pcc: A gpu based technique to compute pairwise pearson's correlation coefficients for big fmri data," in *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, ACM-BCB '17, (New York, NY, USA), pp. 723–728, ACM, 2017.

[20] D. Gembris, M. Neeb, M. Gipp, A. Kugel, and R. Männer, "Correlation analysis on gpu systems using nvidia's cuda," *Journal of Real-Time Image Processing*, vol. 6, pp. 275–280, Dec 2011.

[21] E. Kijsipongse, S. U-ruekolan, C. Ngamphiw, and S. Tongsima, "Efficient large pearson correlation matrix computing using hybrid mpi/cuda," in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 237–241, May 2011.

[22] S. MJ, "Functional magnetic resonance imaging," *The Yale Journal of Biology and Medicine*, vol. 82, pp. 1551–4056, 2009.

[23] S. Ogawa, T. M. Lee, A. R. Kay, and D. W. Tank, "Brain magnetic resonance imaging with contrast dependent on blood oxygenation," *Proceedings of the National Academy of Sciences*, vol. 87, no. 24, pp. 9868–9872, 1990.

[24] C. C. Facility, "Hcp lifespan studies," *Human Connectome Project*, 2017.

[25] C. C. Facility, "Human connectome studies related to human disease," *Human Connectome Project*, 2017.

[26] U. of Minnesota, "Human connectome project: Mapping structural and functional connections in the human brain," *Human Connectome Project*, 2015.

[27] S. M. Smith, "The future of fmri connectivity," *NeuroImage*, vol. 62, no. 2, pp. 1257 – 1266, 2012. 20 YEARS OF fMRI.

[28] S. Whitfield-Gabrieli and A. Nieto-Castanon, "Conn: A functional connectivity toolbox for correlated and anticorrelated brain networks," *Brain Connectivity*, vol. 2, no. 3, pp. 125–141, 2012. PMID: 22642651.

[29] C. F. C. Toolbox, "Conn functional connectivity toolbox," 2018.

[30] Xilinx, "Axi reference guide - ug761," 03 2011.

[31] N. N. I. T. Initiative, "Nifti: Neuroimaging informatics technology initiative," 2016.

[32] P. Hagmann, L. Cammoun, X. Gigandet, R. Meuli, C. J. Honey, V. J. Wedeen, and O. Sporns, "Mapping the structural core of human cerebral cortex," *PLOS Biology*, vol. 6, pp. 1–15, 07 2008.

[33] Xilinx, "Kcu105 eval board," 03 2018.

[34] Xilinx, "Kintex ultrascale+ product advantage," 03 2018.

[35] Avnet, "Picozed," 03 2018.

[36] Altium, "Altium documentaion," 03 2018.

[37] S. D. Assocation, "Sd standard overview," 03 2018.

[38] T. Instruments, "Lvds application and data handbook," 11 2002.

[39] Xilinx, "Zynq-7000 all programmable soc first generation architecture," 06 2017.

[40] Y. Wang, H. Du, M. Xia, L. Ren, M. Xu, T. Xie, G. Gong, N. Xu, H. Yang, and Y. He, "A hybrid cpu-gpu accelerated framework for fast mapping of high-resolution human brain connectome," *PLOS ONE*, vol. 8, pp. 1–14, 05 2013.

[41] . F. C. Project, “1000 functional connectomes project,” 2018.

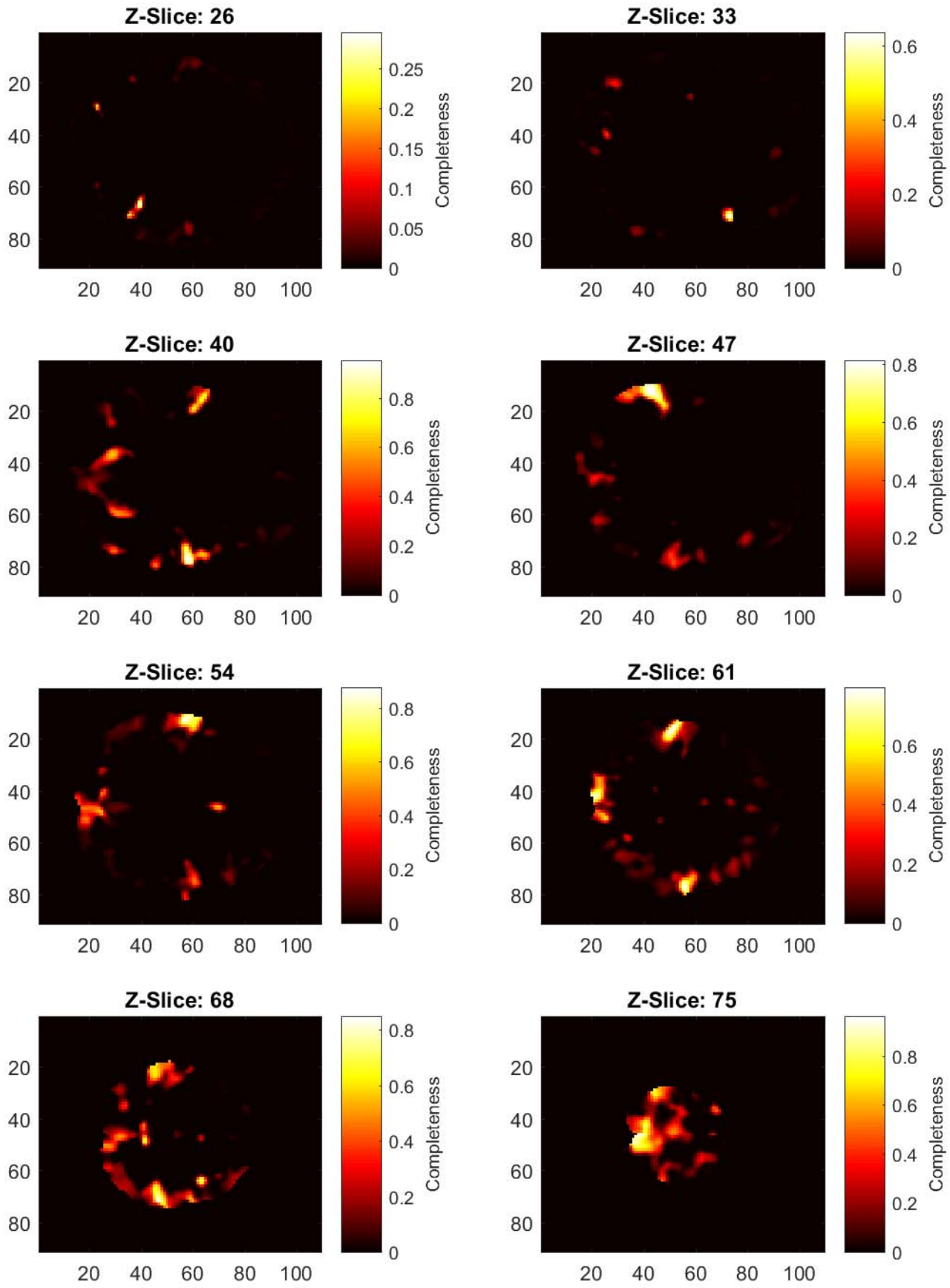
[42] D. Kennedy, C. Haselgrove, *et al.*, “Harvard-oxford atlas,” 03 2018.

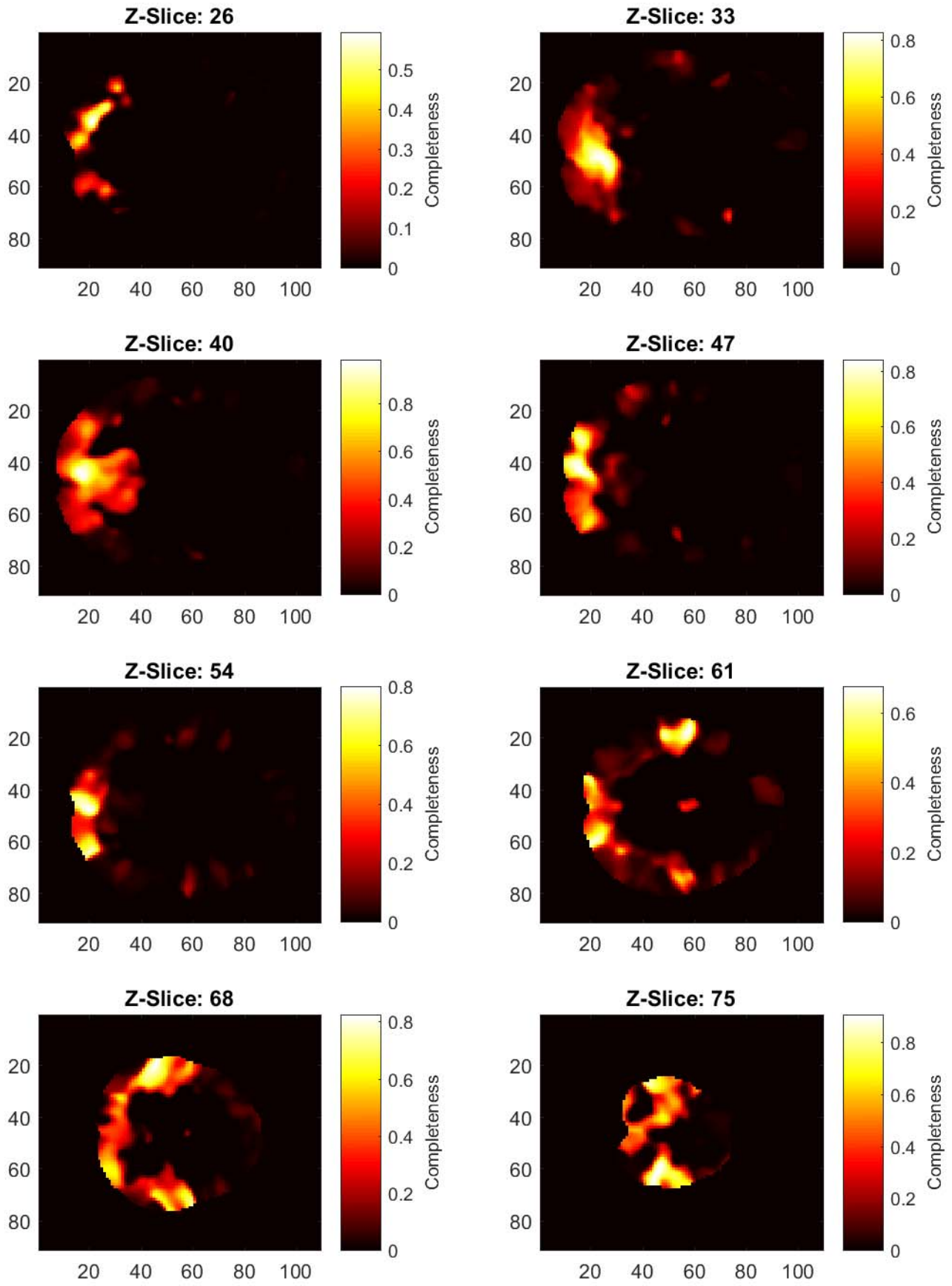
[43] A. Nieto-Castanon, “Forum post: Global correlation positive and negative r?,” 03 2017.

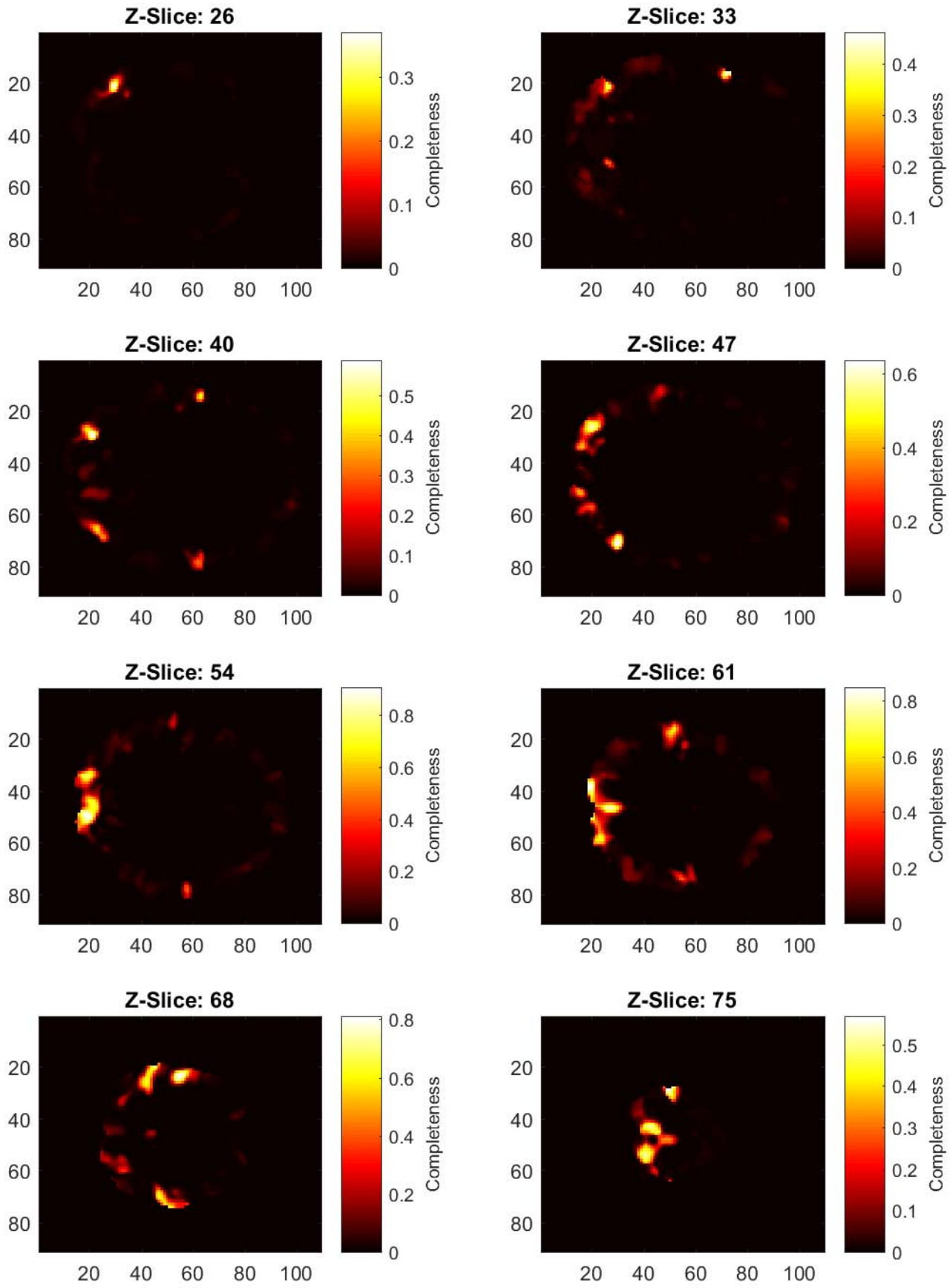
[44] A. Nieto-Castanon, “Forum post: Extracting correlation matrix from conn,” 12 2013.

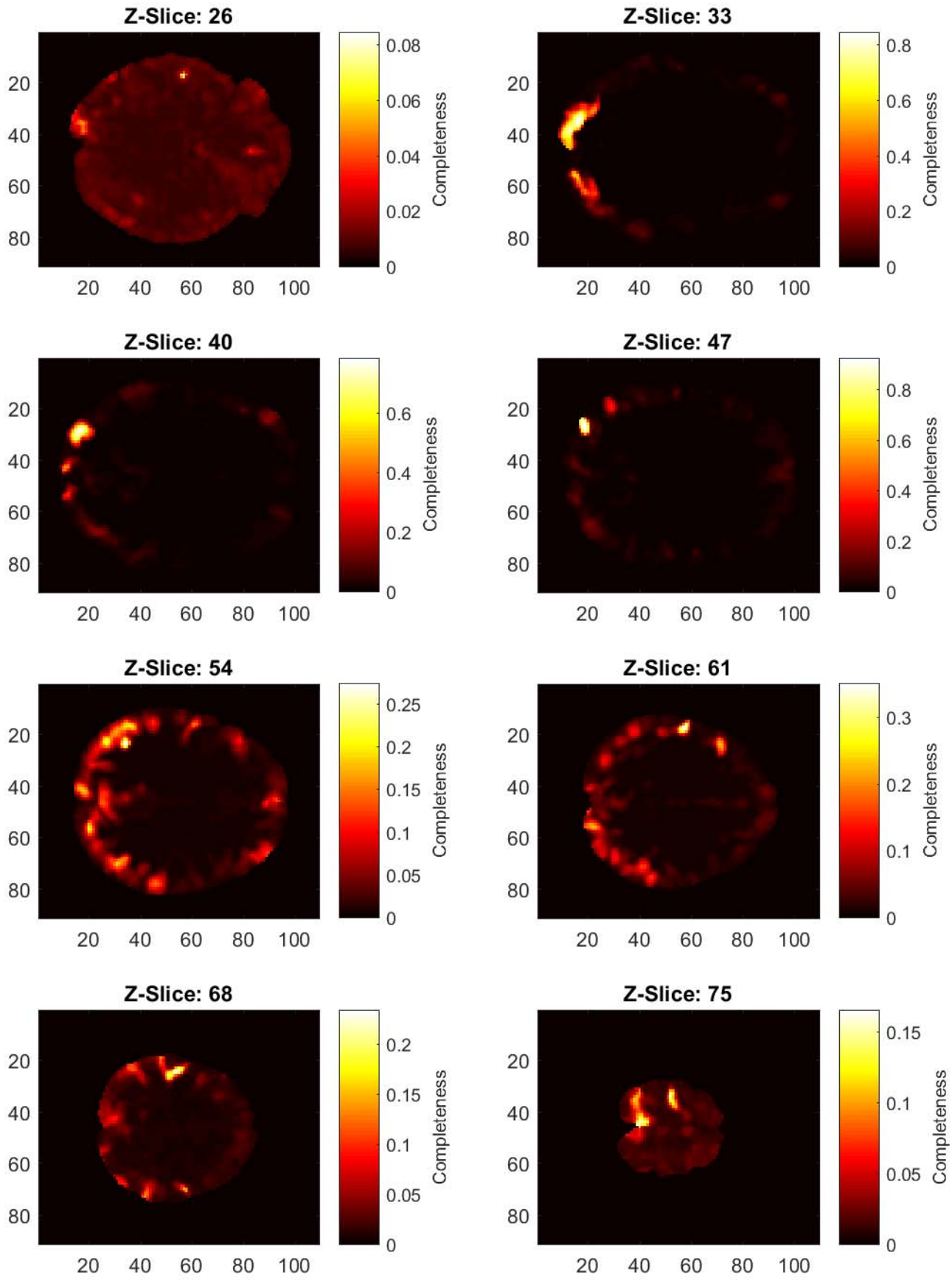
APPENDIX A

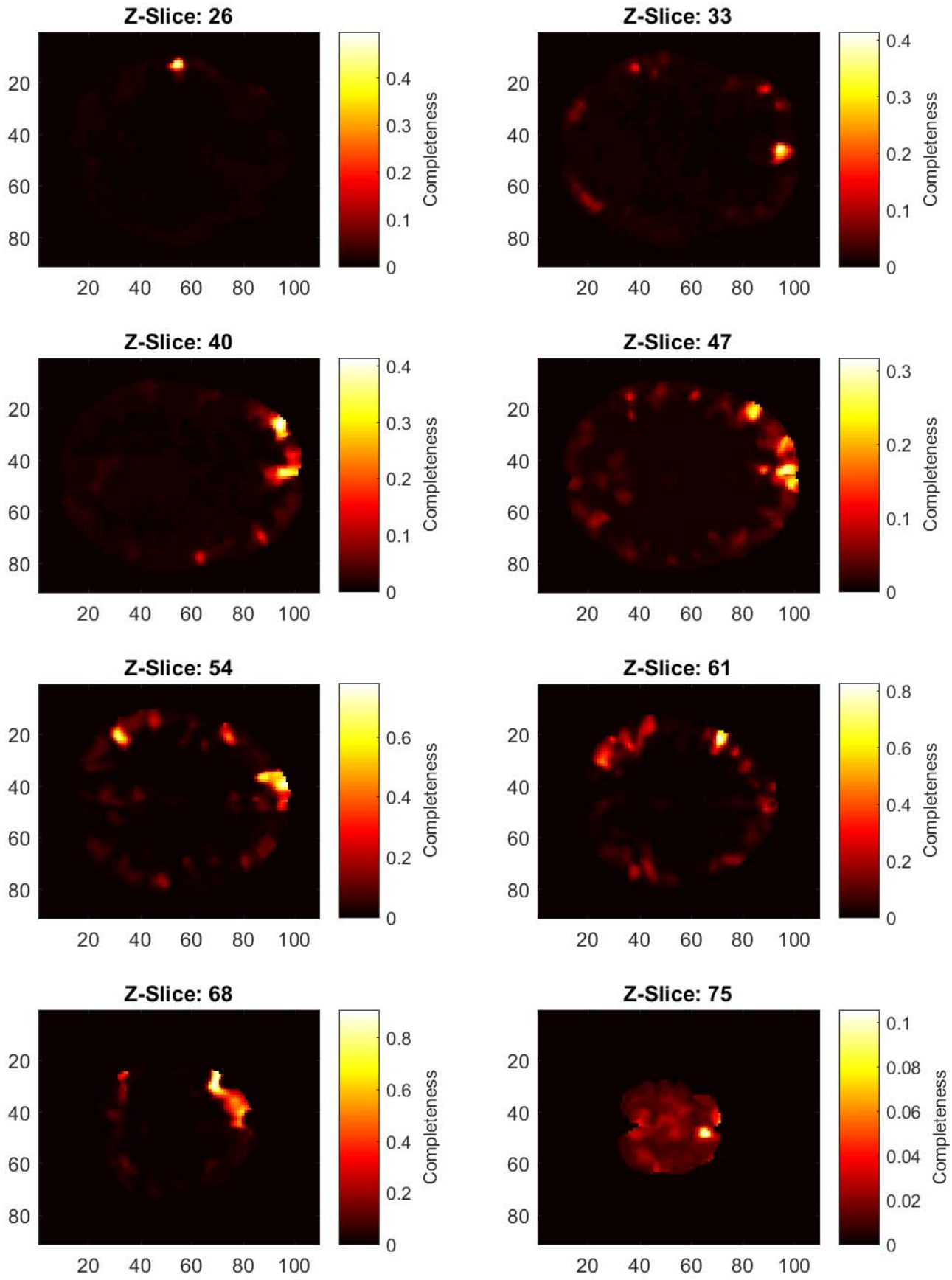
DETAILED RESULTS

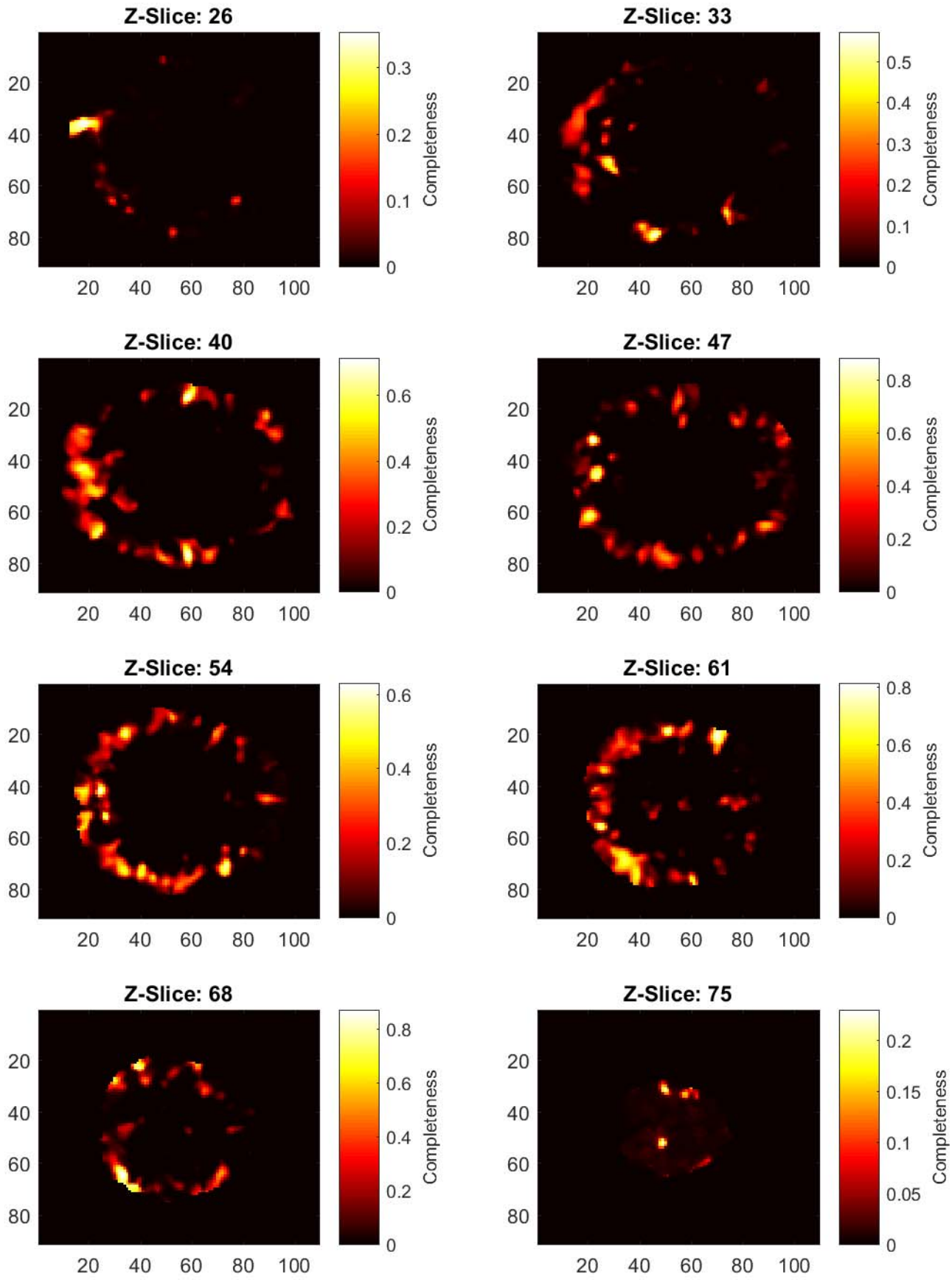

Connectedness of HCP Subject: 102816
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

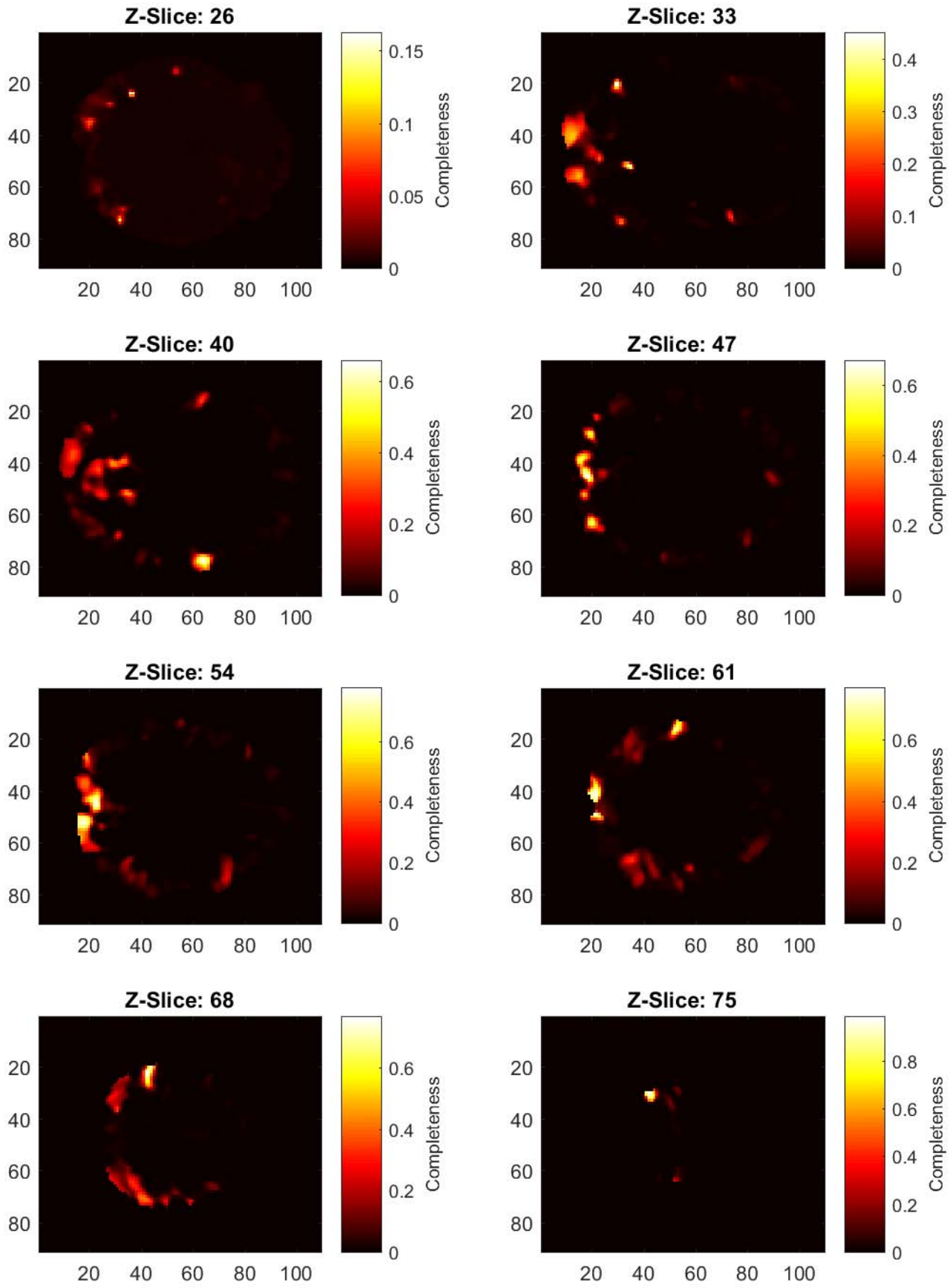

Connectedness of HCP Subject: 103818
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

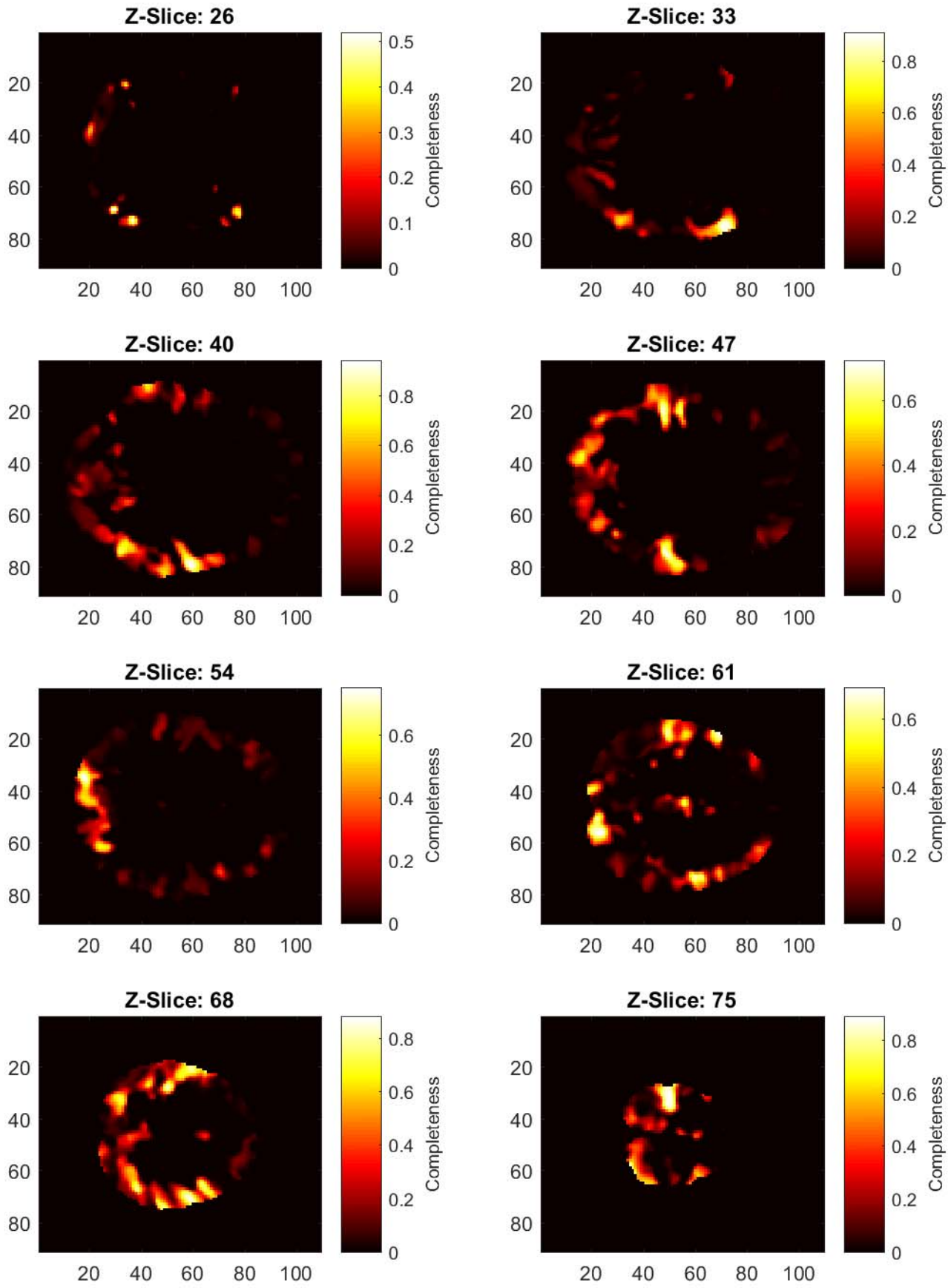

Connectedness of HCP Subject: 104416
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

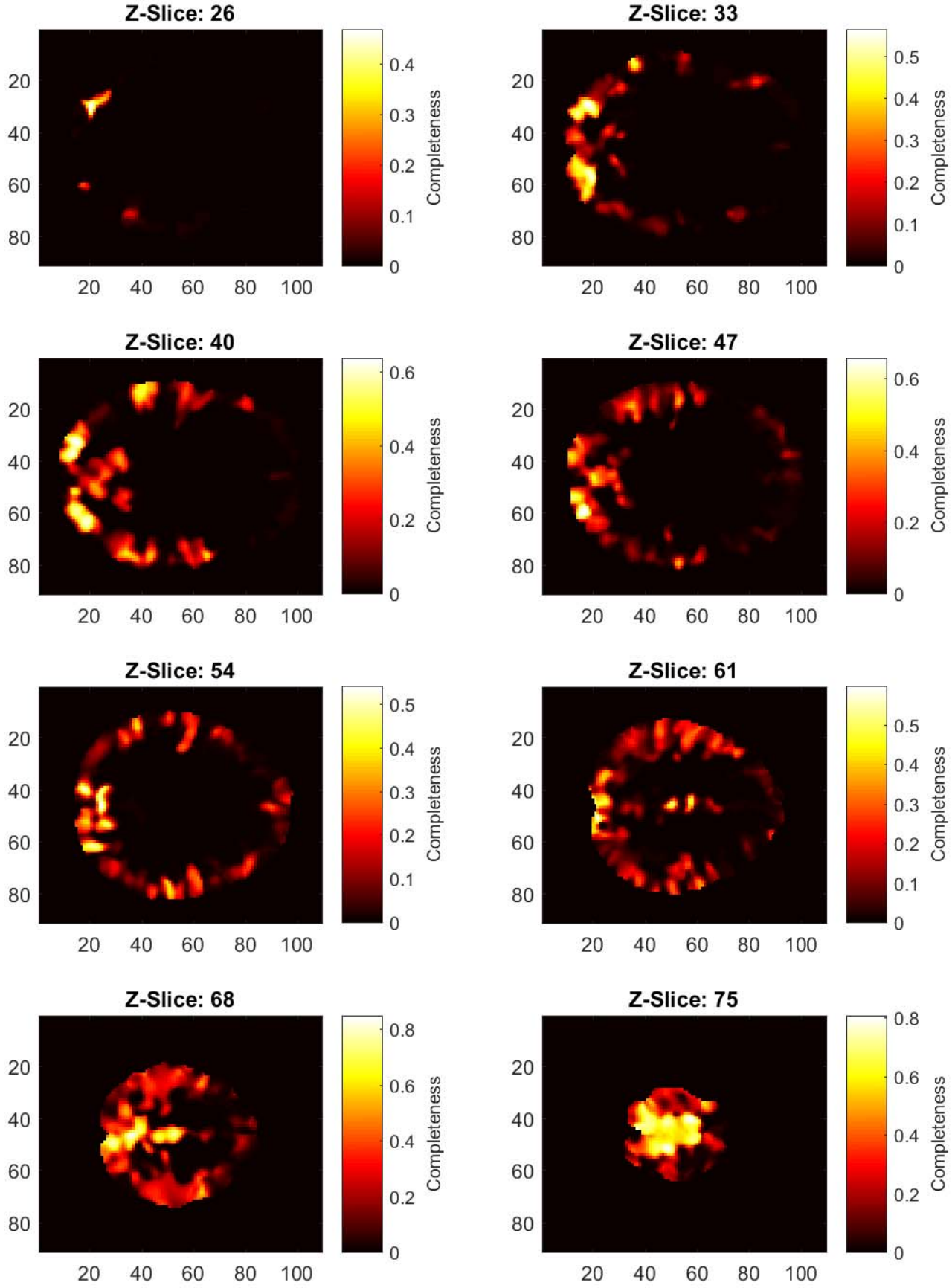

Connectedness of HCP Subject: 105014
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

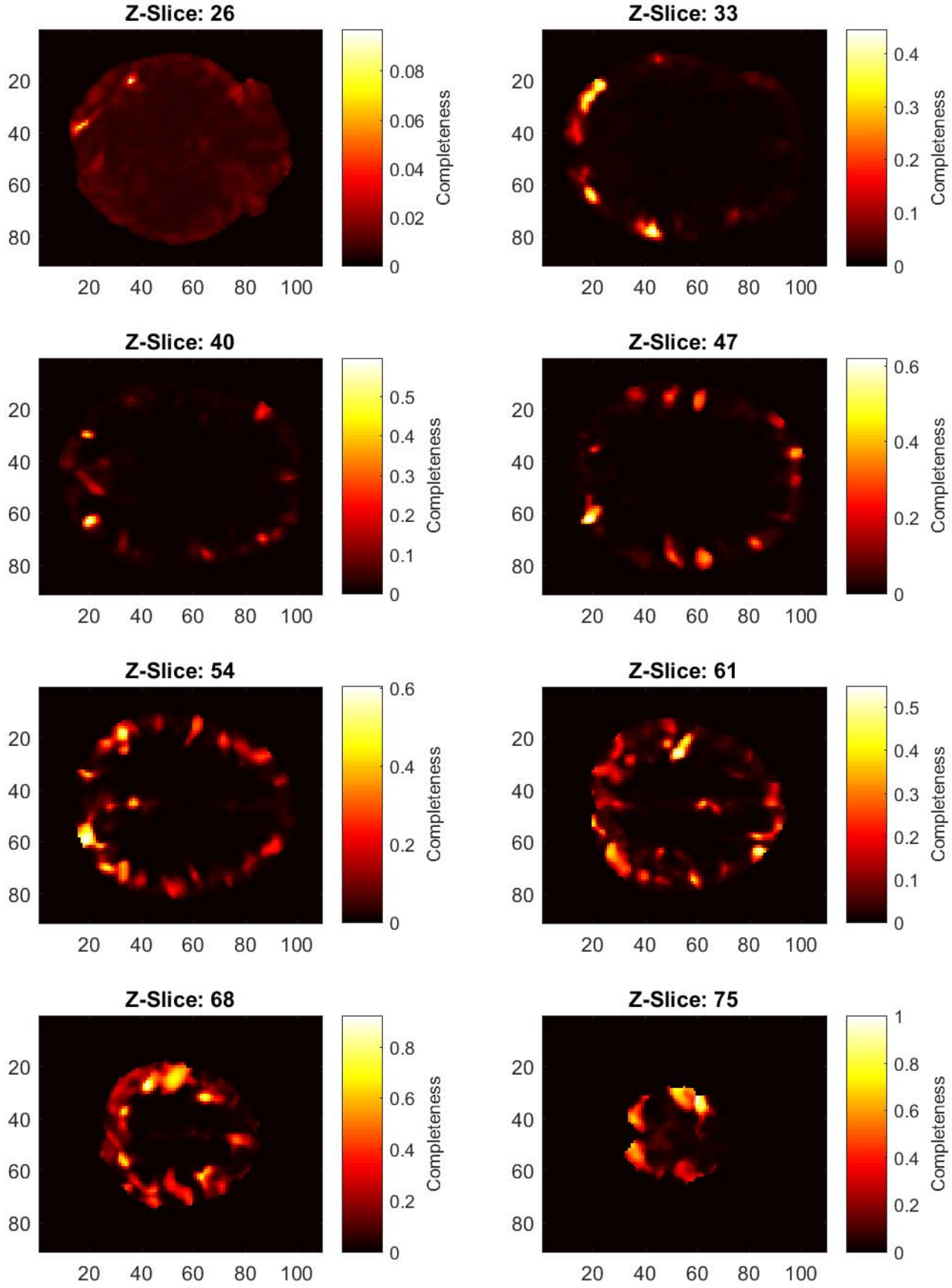

Connectedness of HCP Subject: 105115
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

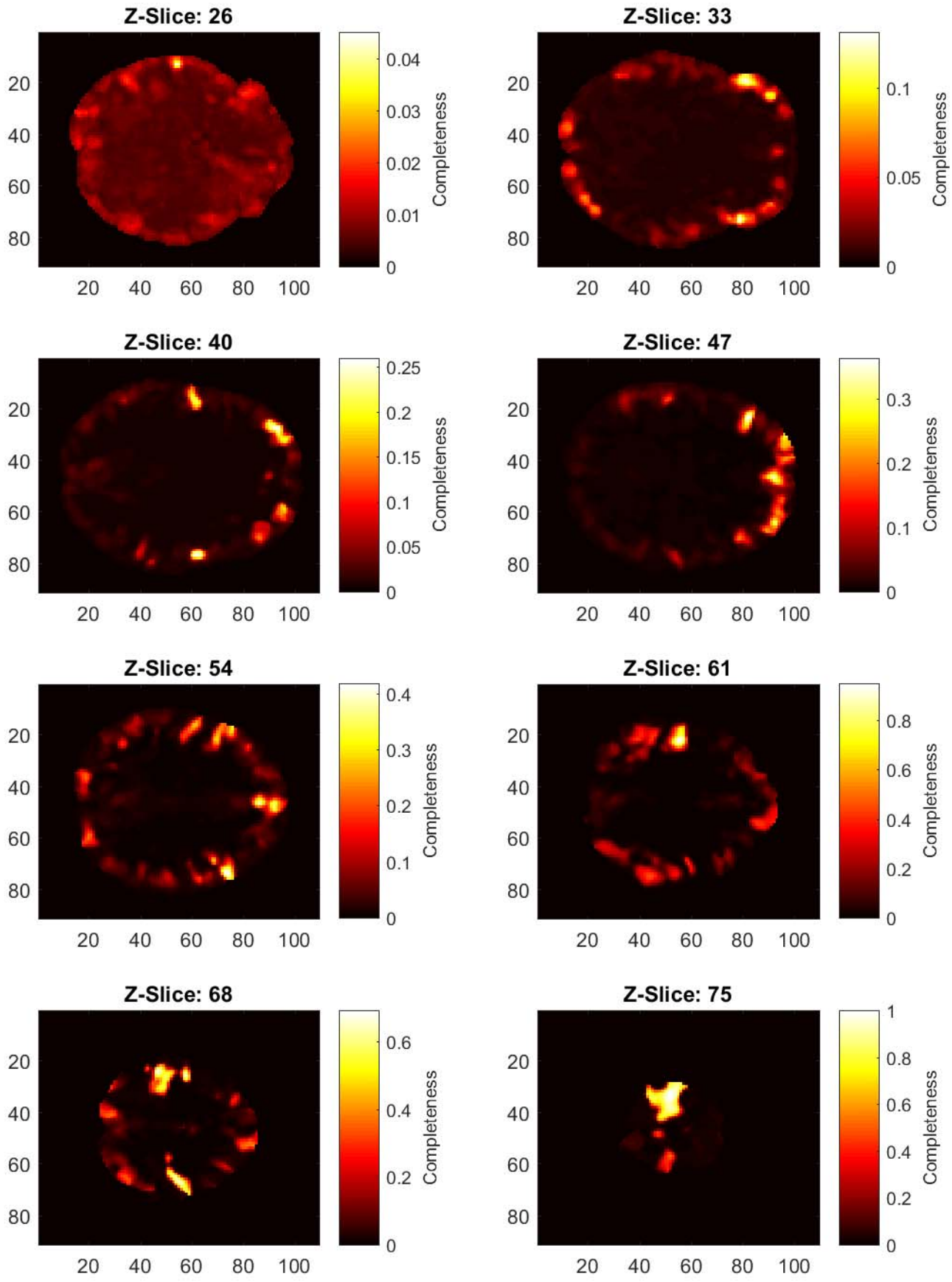

Connectedness of HCP Subject: 105923
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

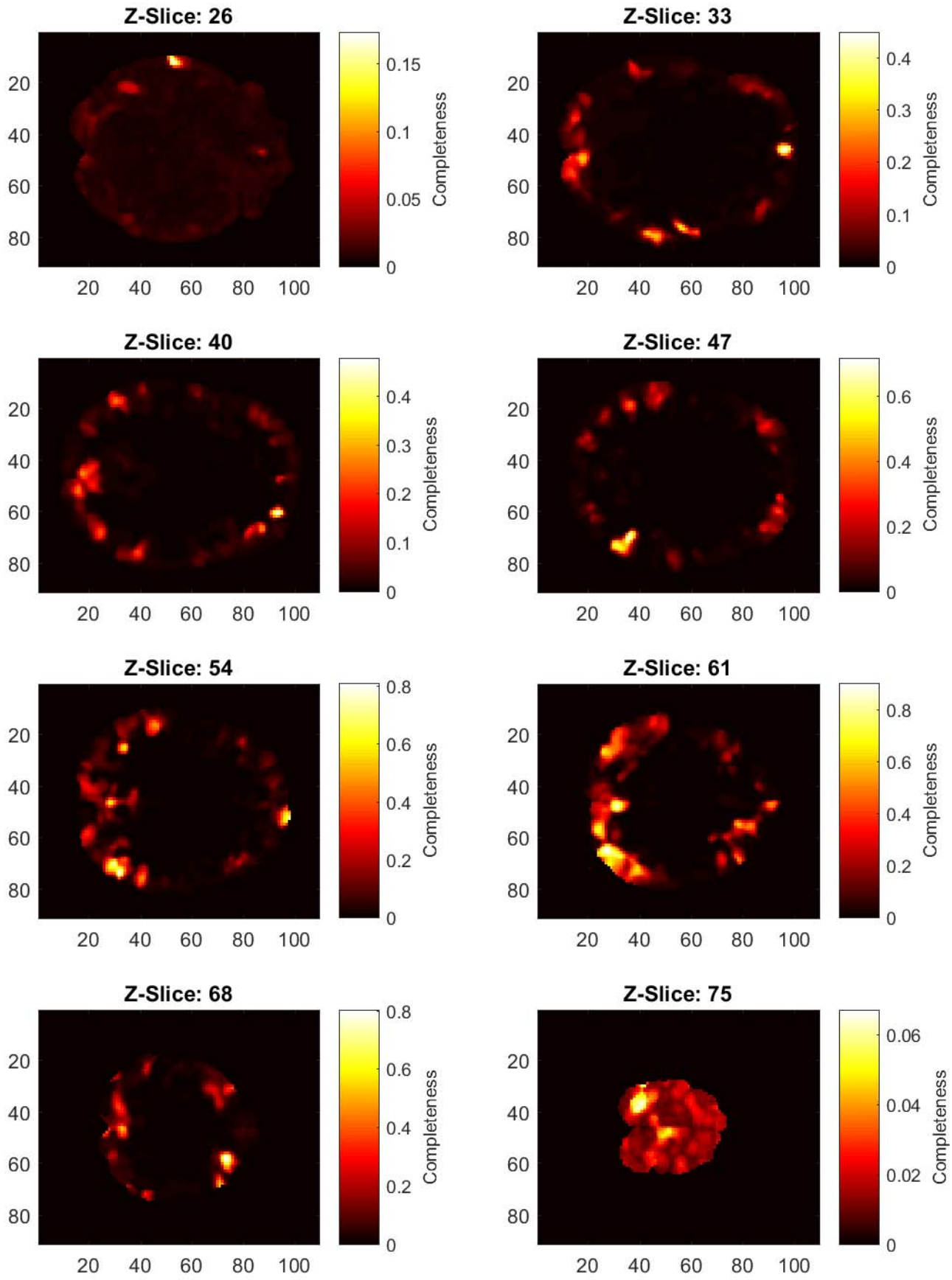

Connectedness of HCP Subject: 106016
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

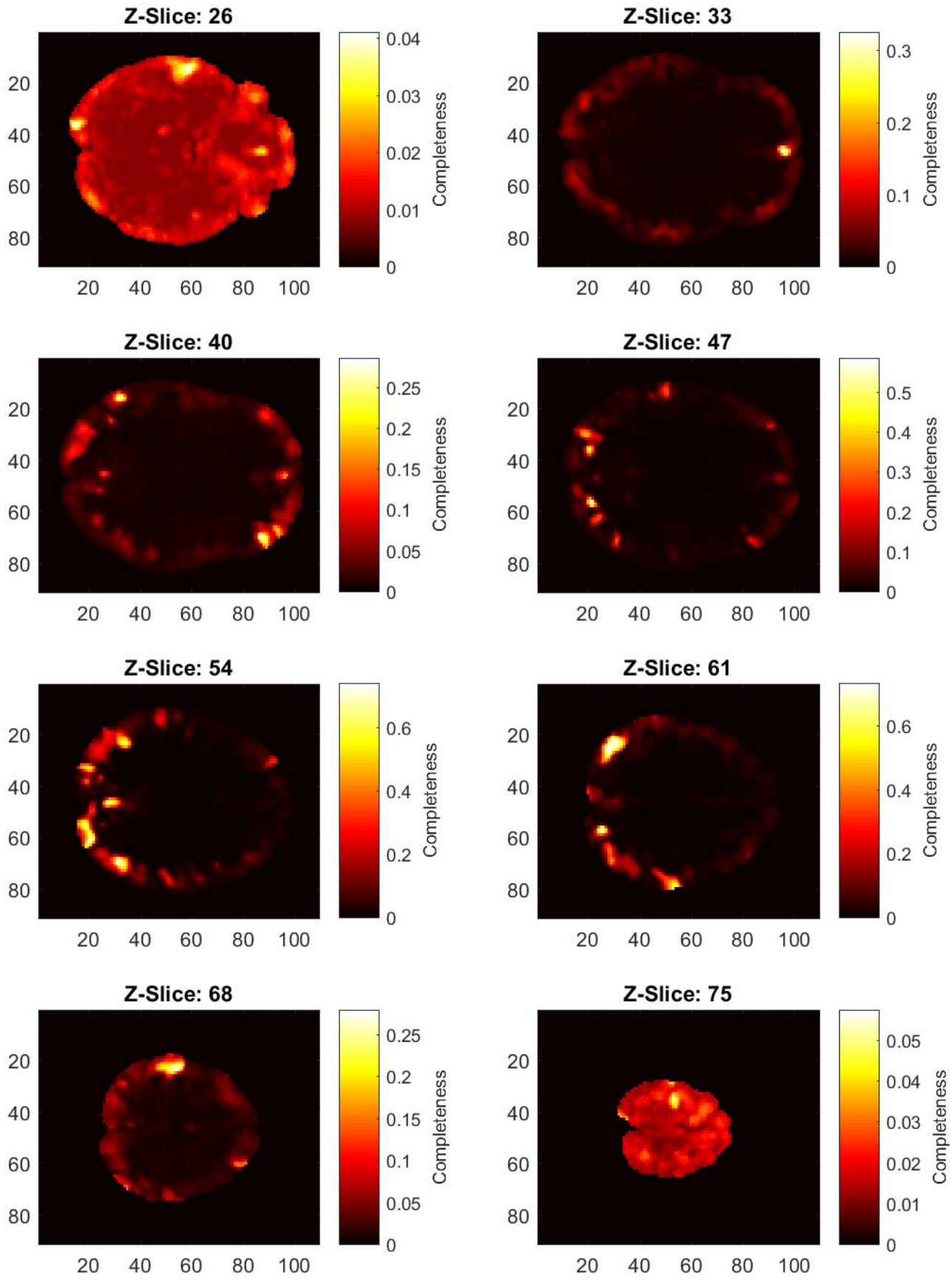

Connectedness of HCP Subject: 107422
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

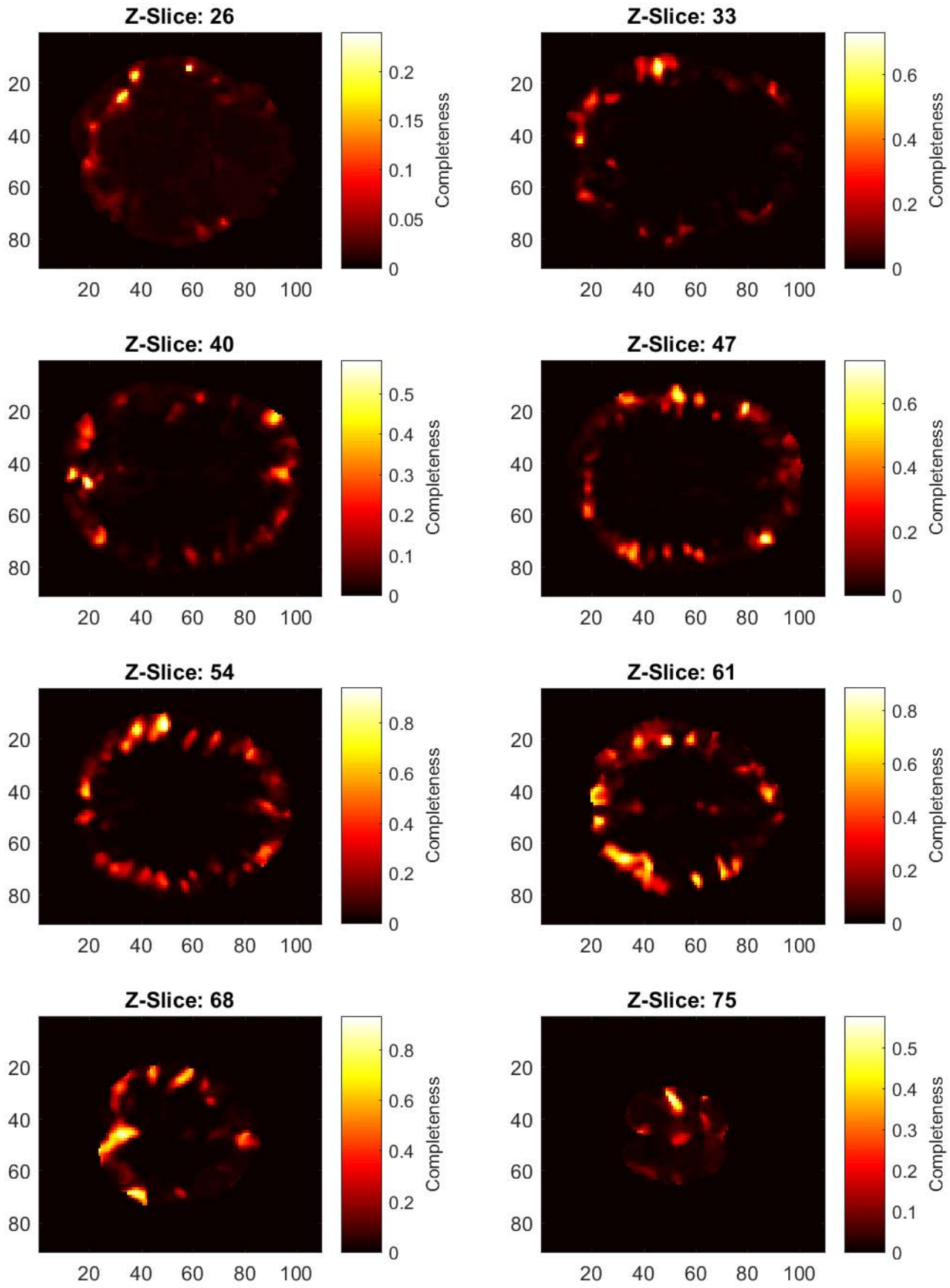

Connectedness of HCP Subject: 108525
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

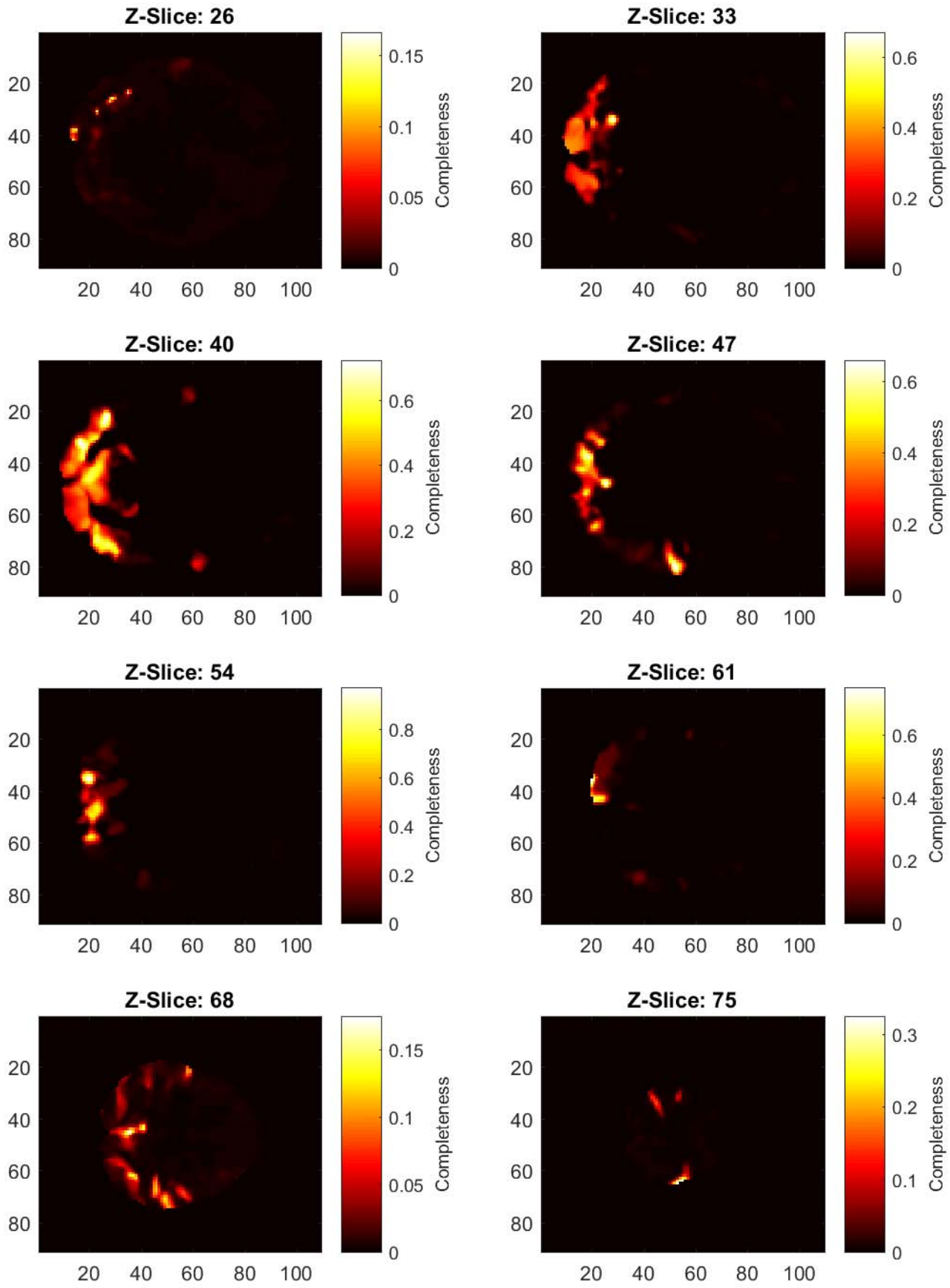

Connectedness of HCP Subject: 109325
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

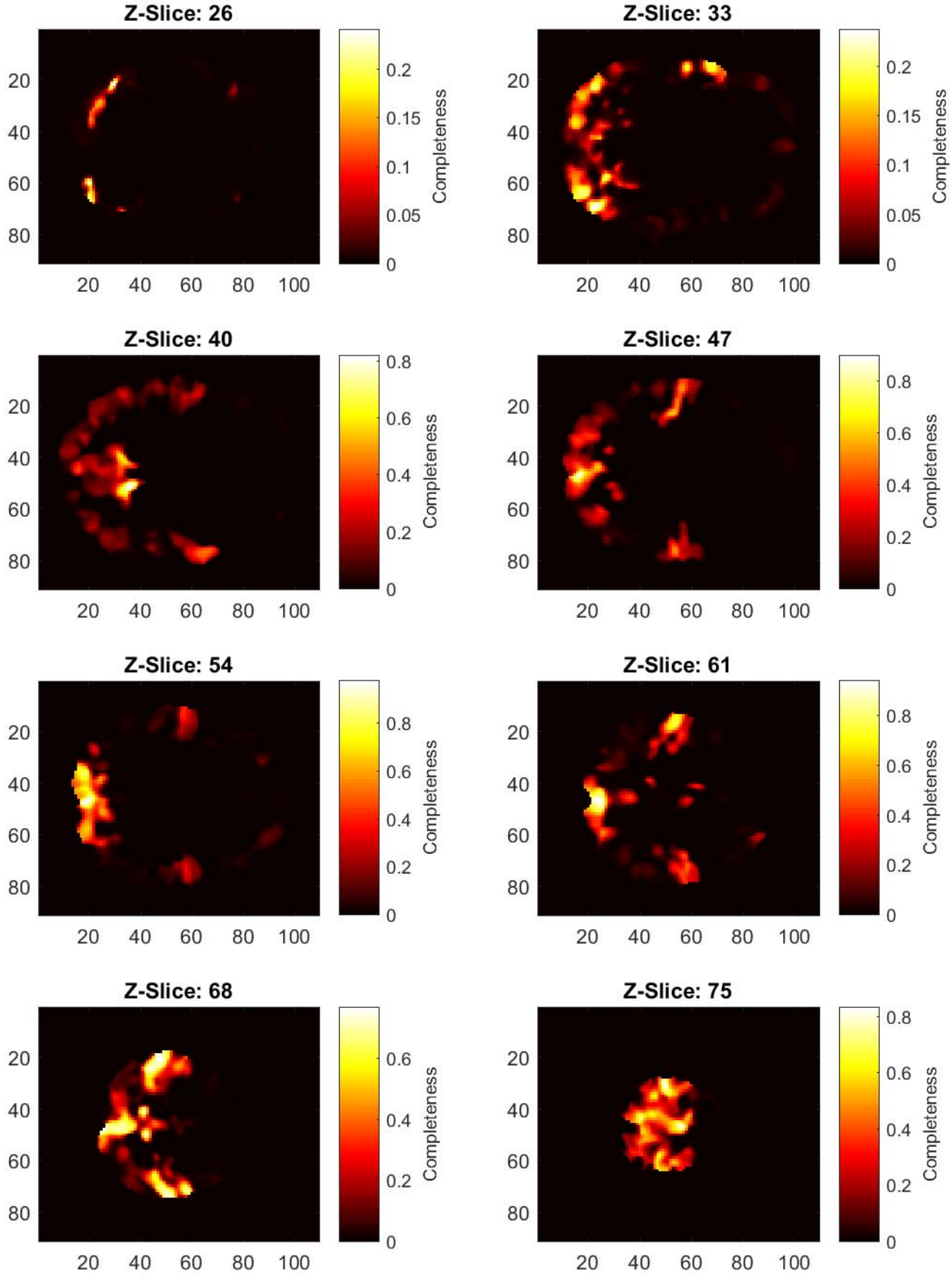

Connectedness of HCP Subject: 111009
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

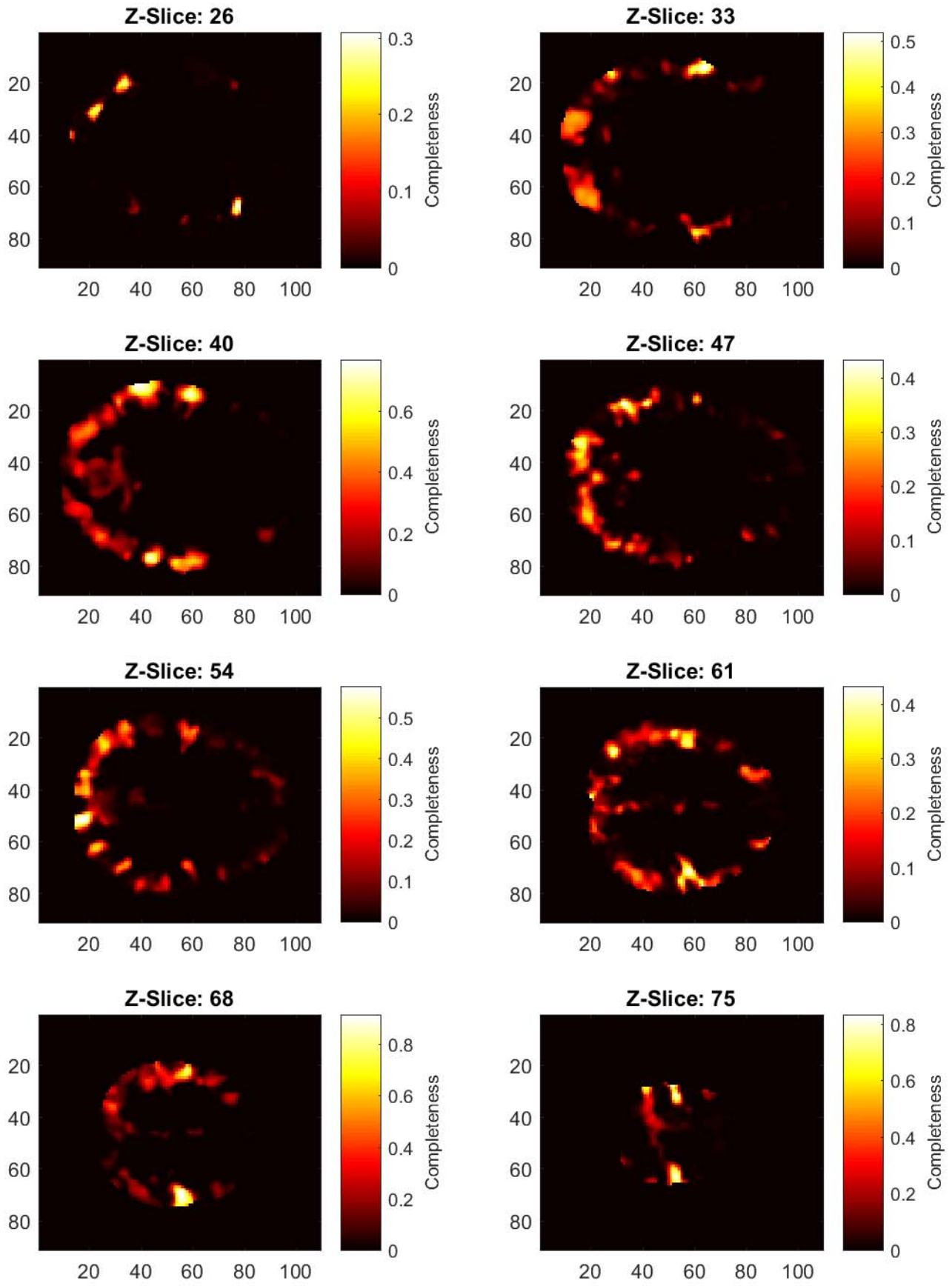

Connectedness of HCP Subject: 111413
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

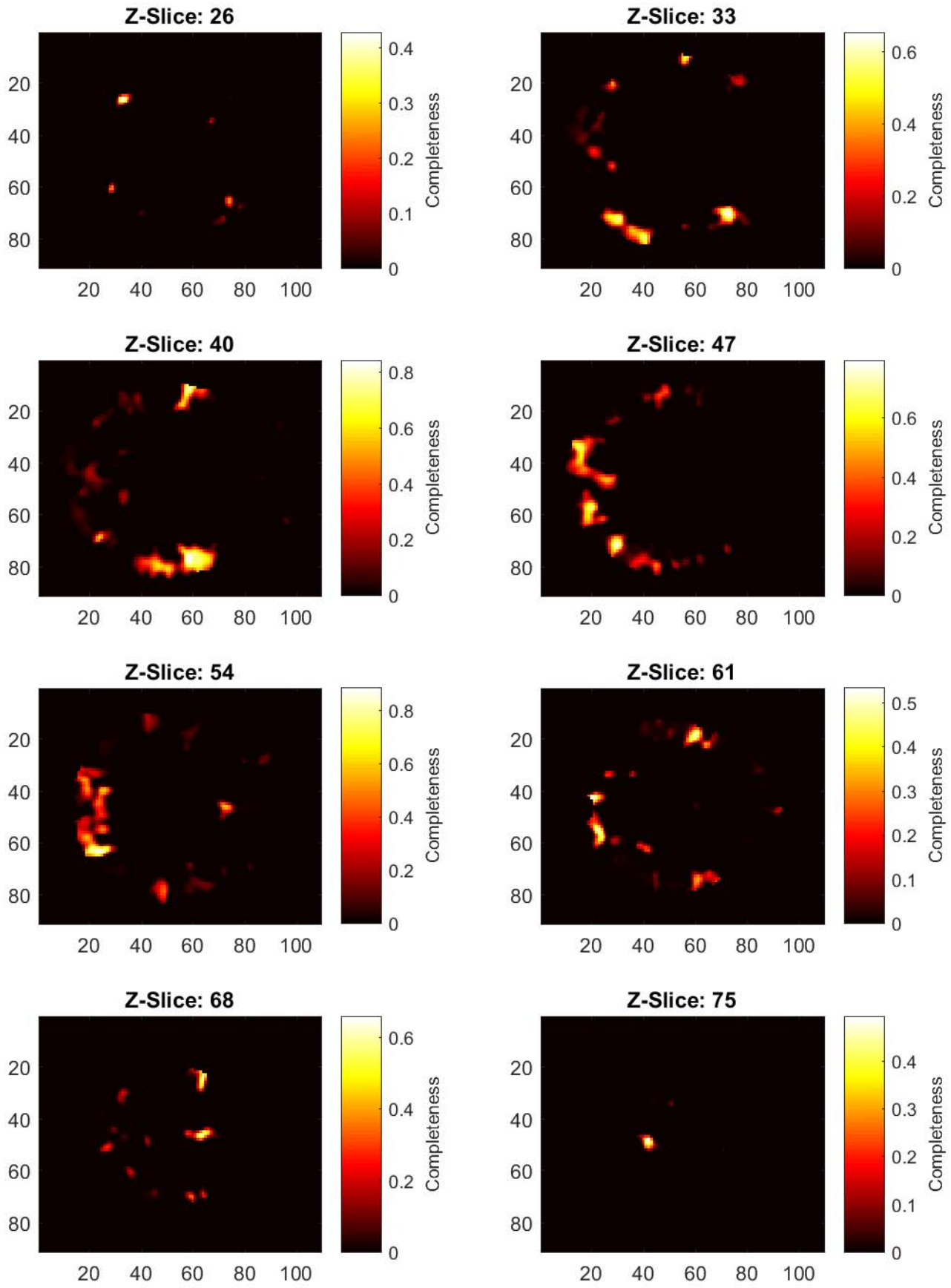

Connectedness of HCP Subject: 111716
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

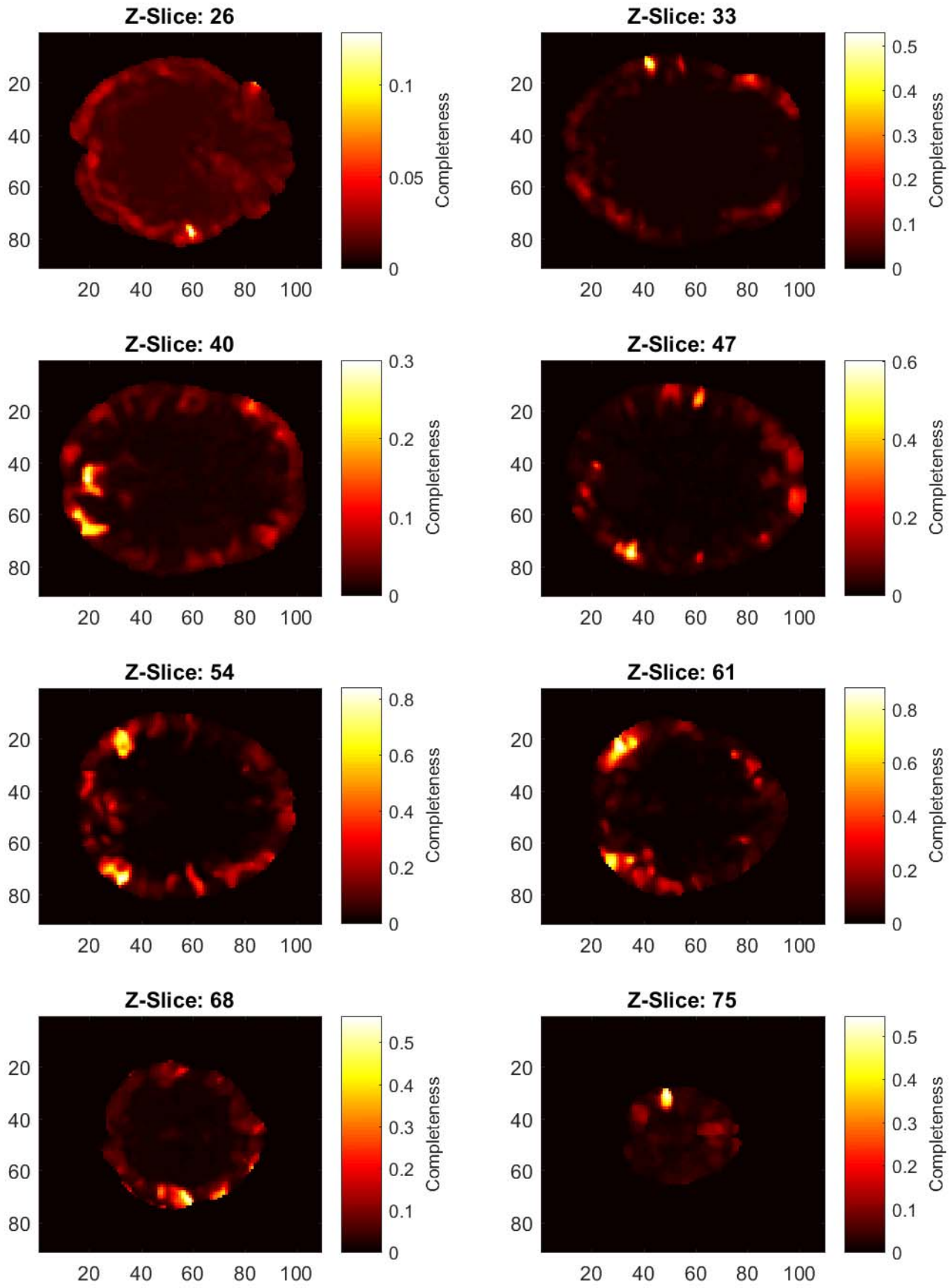

Connectedness of HCP Subject: 112314
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

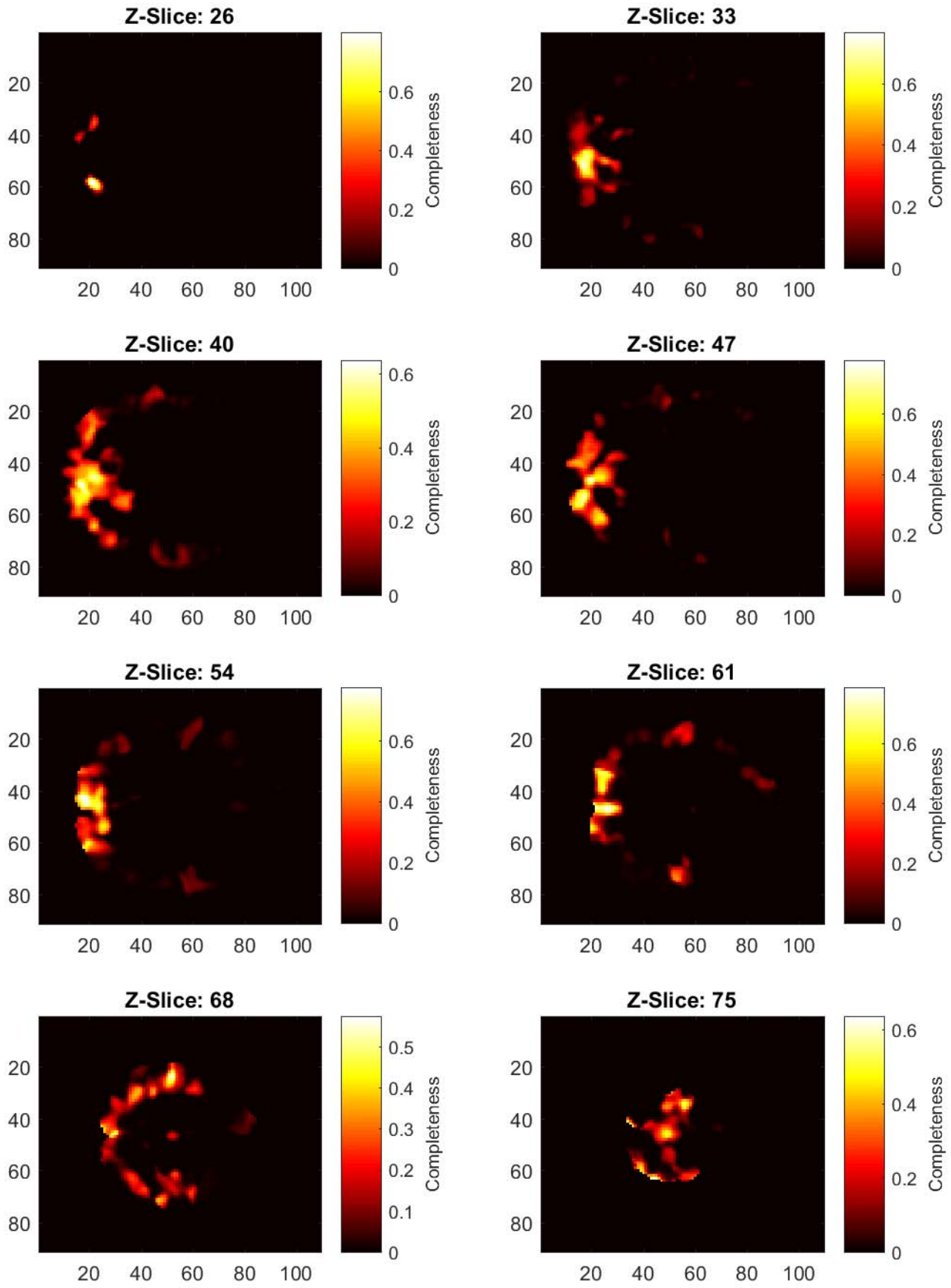

Connectedness of HCP Subject: 112516
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

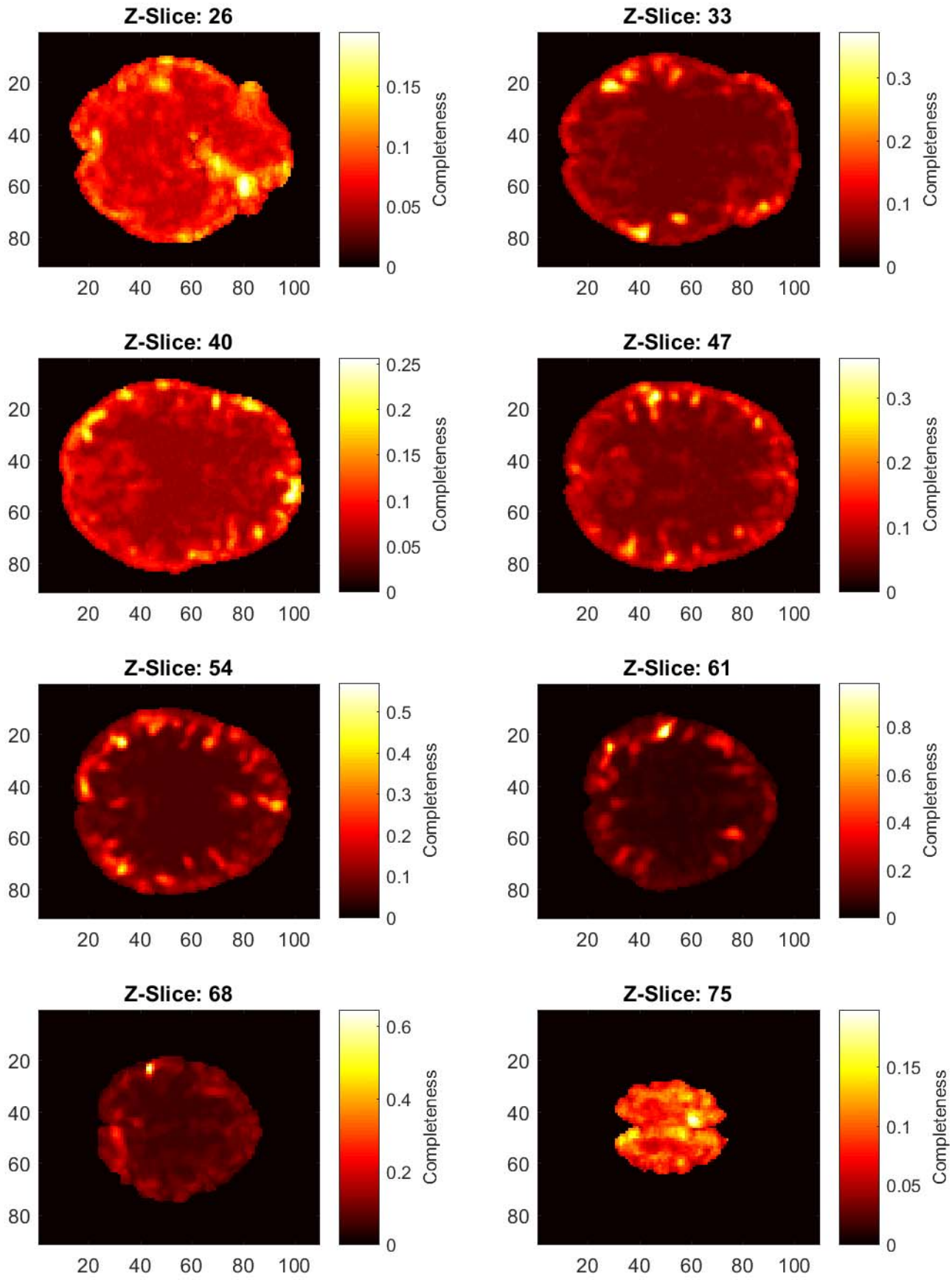

Connectedness of HCP Subject: 113215
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100
0
0.05
0.1
0.15
0.2
0.4
0.6
0.8

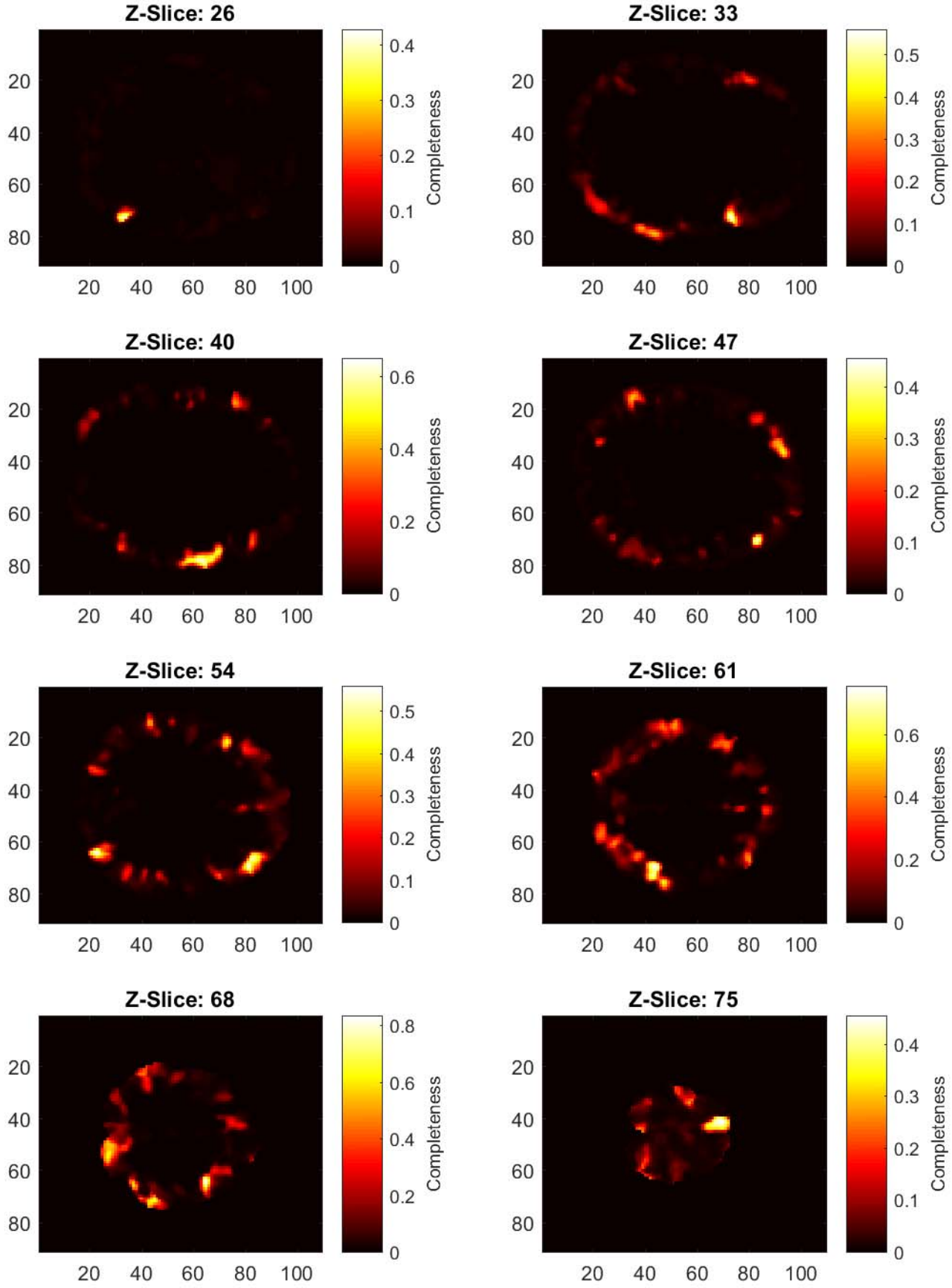

Connectedness of HCP Subject: 113922
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

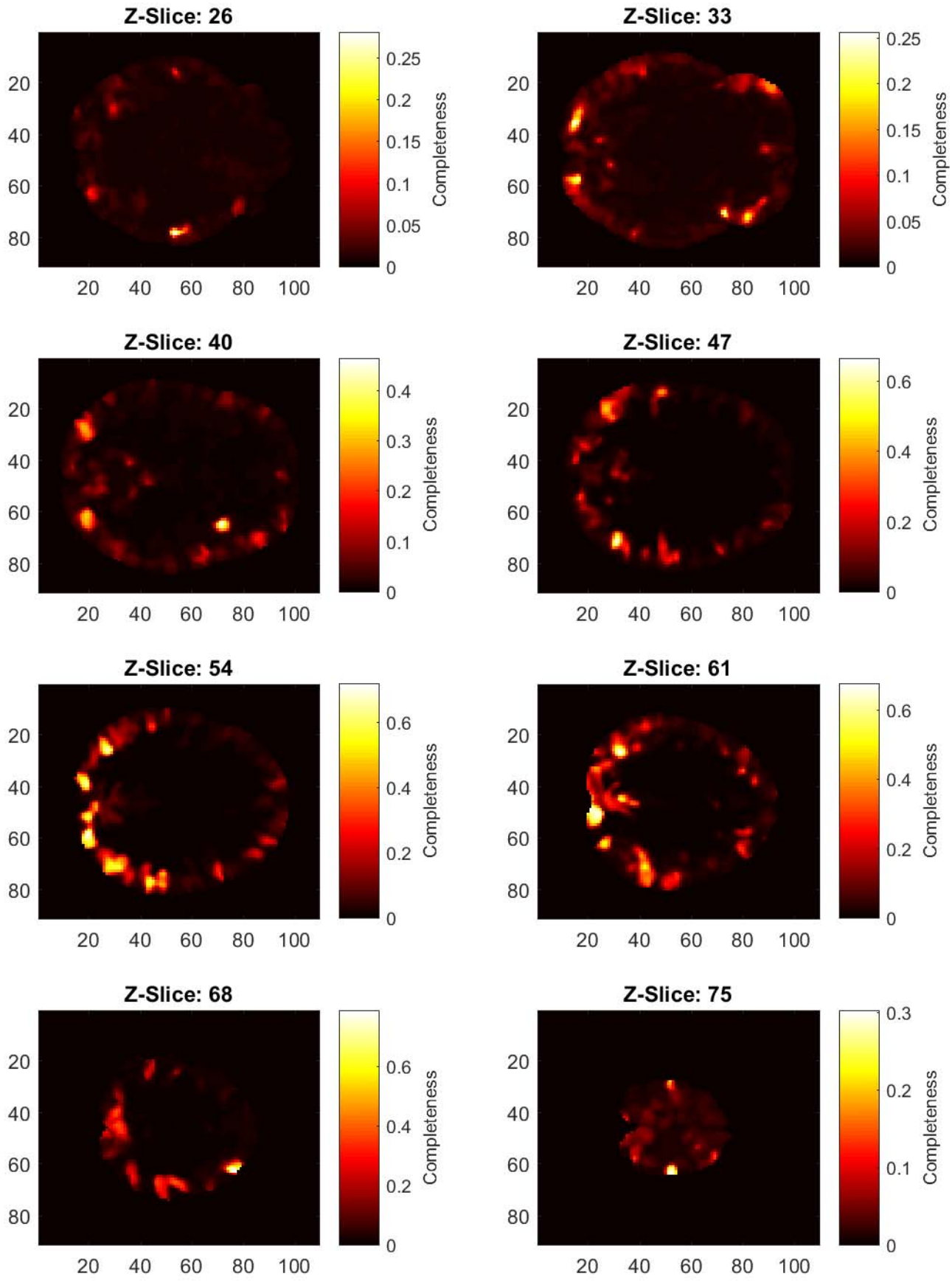

Connectedness of HCP Subject: 114924
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

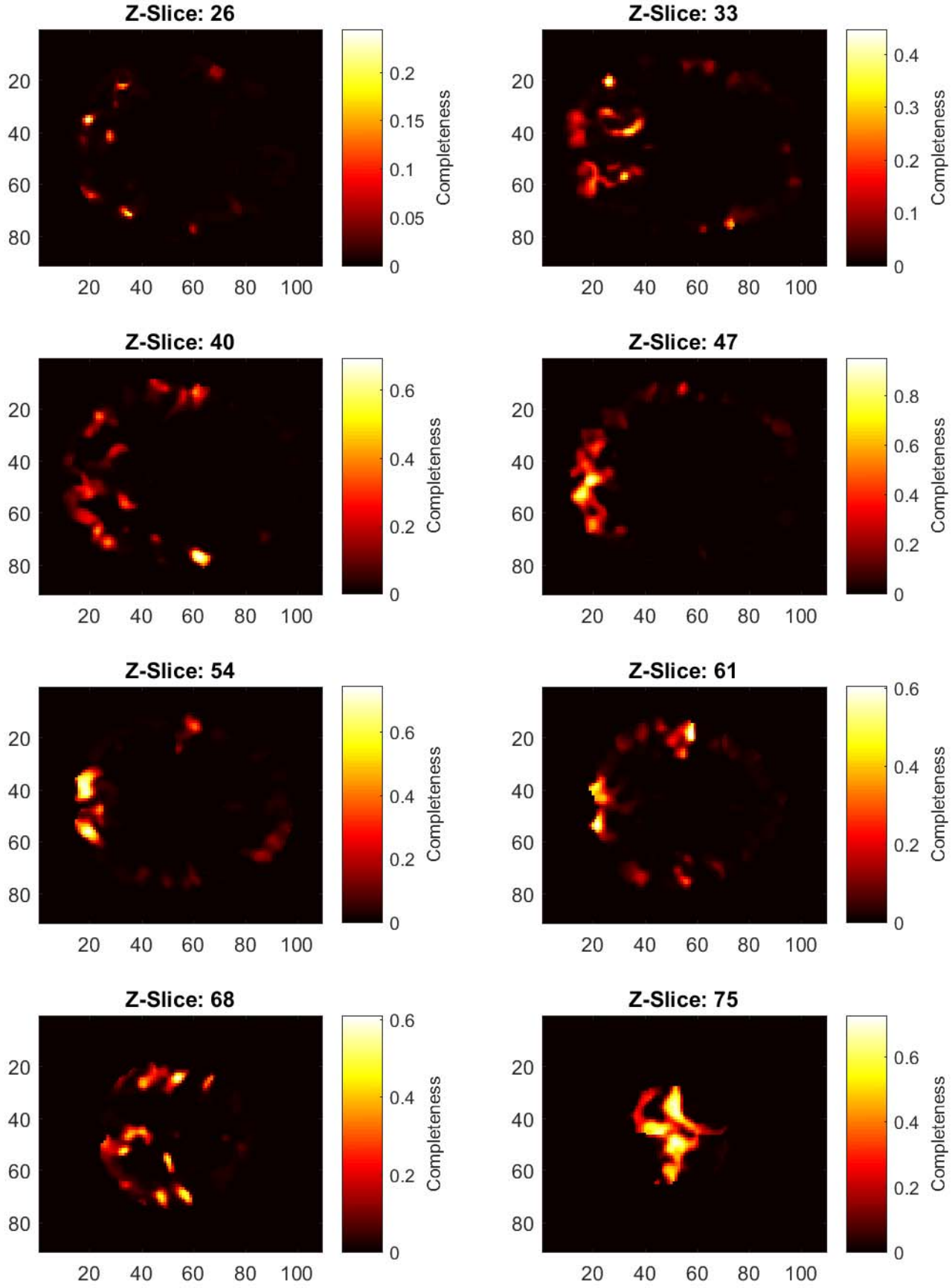

Connectedness of HCP Subject: 116726
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100


Connectedness of HCP Subject: 117122
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100
0
0.1
0.2
0.3
0.4
0.5
0.6


Connectedness of HCP Subject: 118124
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

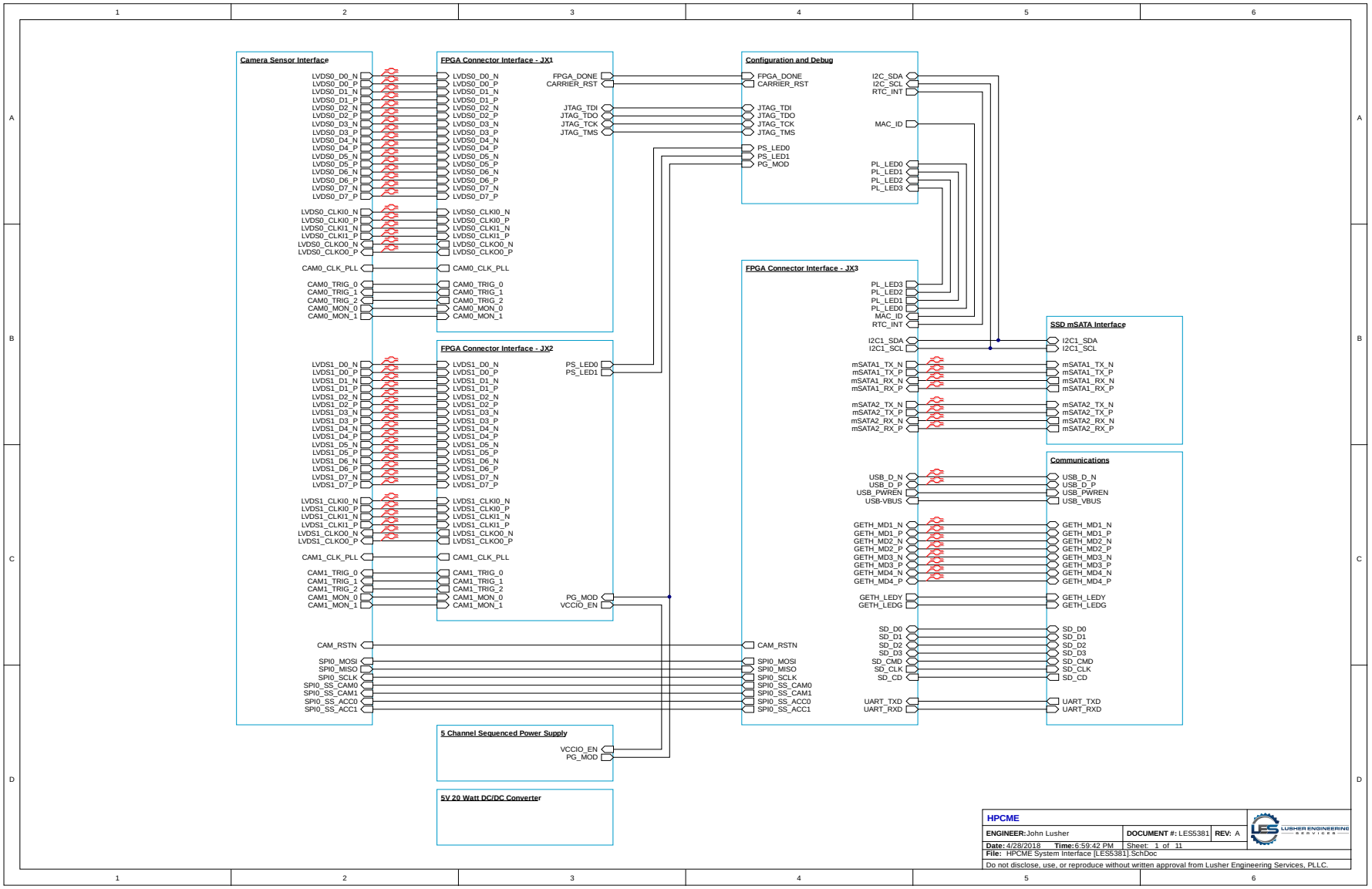

Connectedness of HCP Subject: 118225
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100


Connectedness of HCP Subject: 119126
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

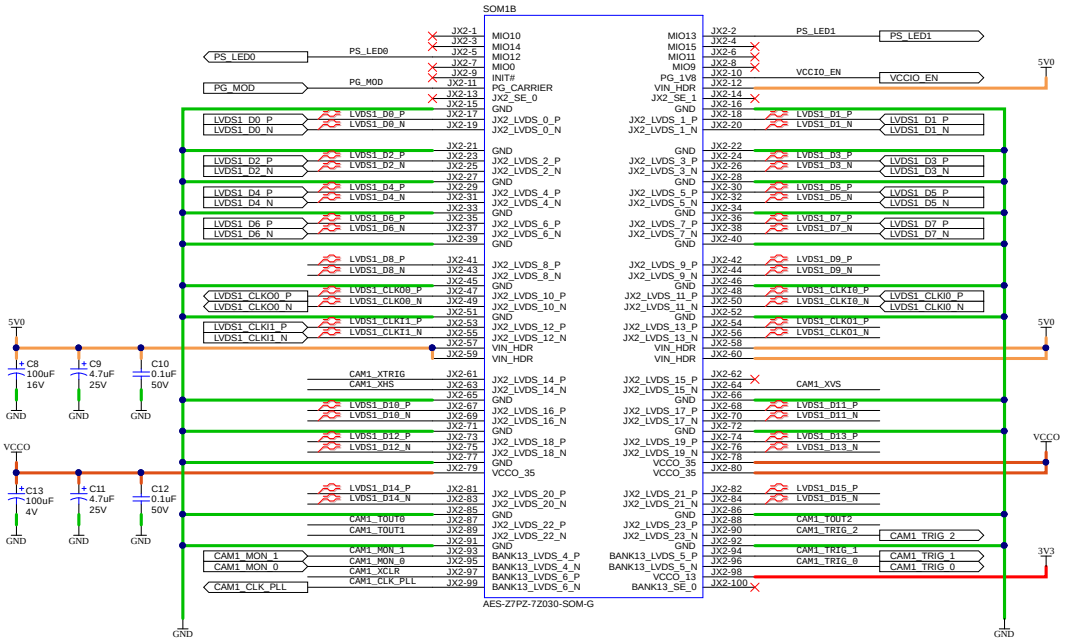

Connectedness of HCP Subject: 119833
Z-Slice: 26
Z-Slice: 33
Z-Slice: 40
Z-Slice: 47
Z-Slice: 54
Z-Slice: 61
Z-Slice: 68
Z-Slice: 75
Completeness
20
40
60
80
100

APPENDIX B

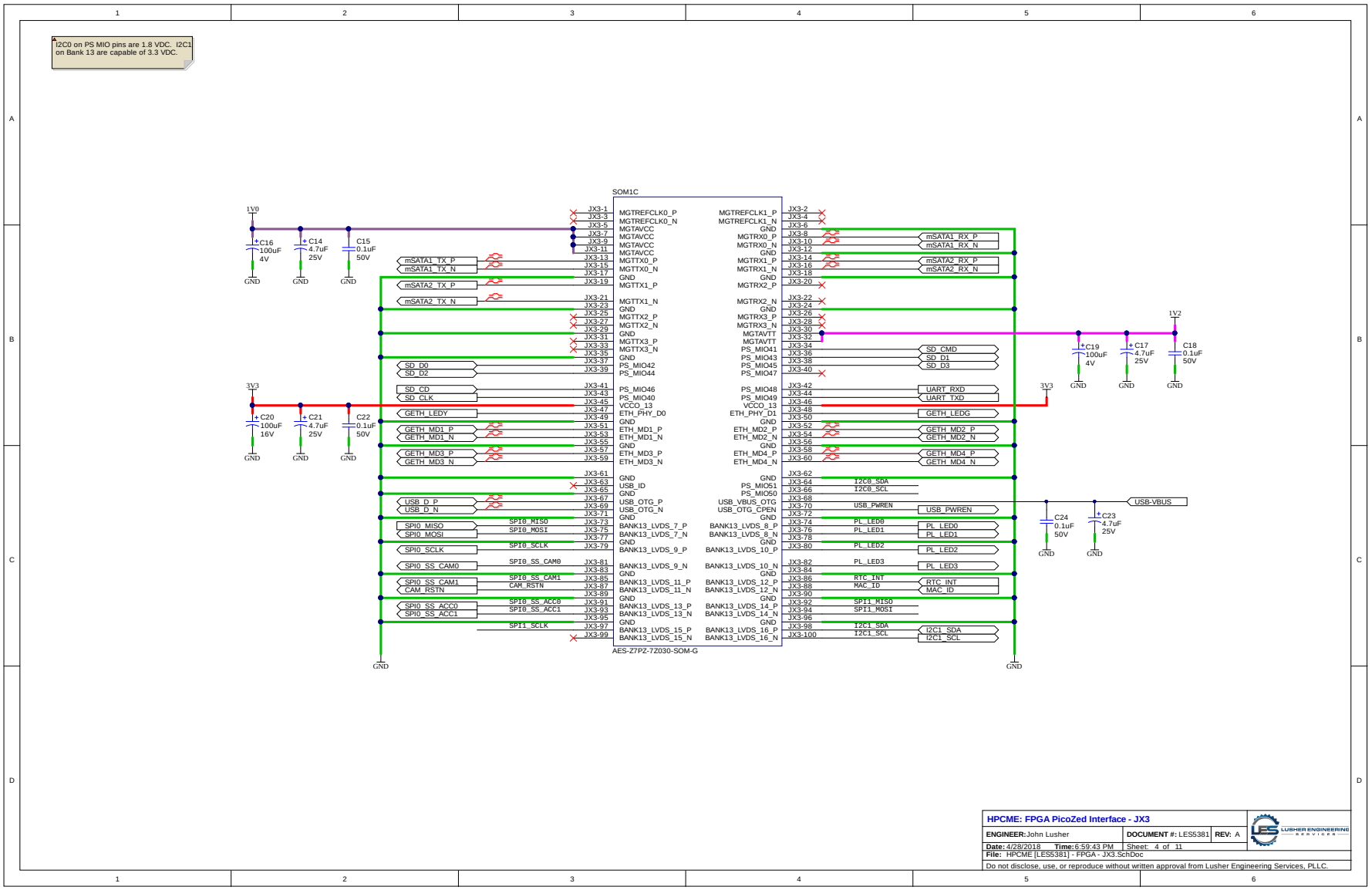
NODE DEGREE ENGINE - SCHEMATICS AND PCB LAYOUT

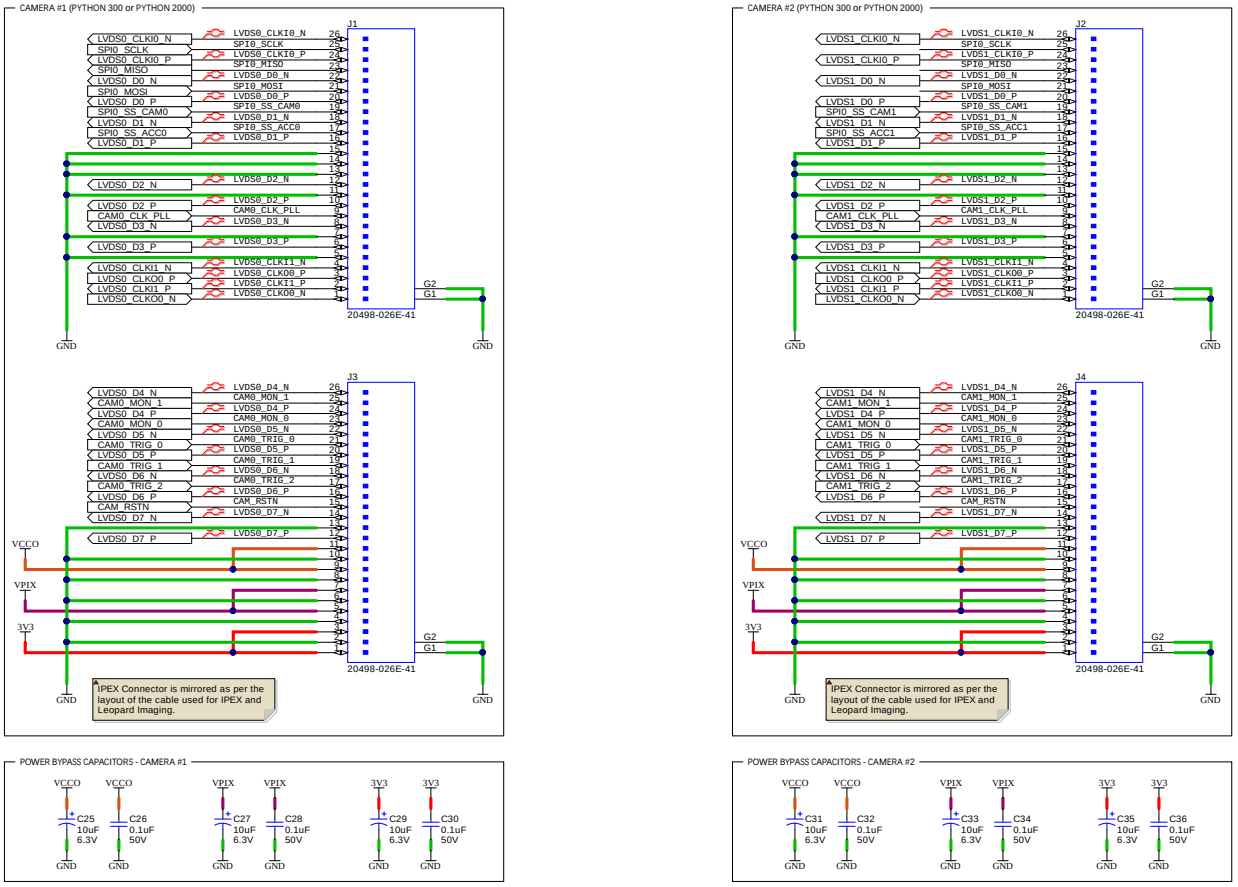

Camera Sensor Interface
LVDS0_D0_N
LVDS0_D0_P
LVDS0_D1_N
LVDS0_D1_P
LVDS0_D2_N
LVDS0_D2_P
LVDS0_D3_N
LVDS0_D3_P
LVDS0_D4_N
LVDS0_D4_P
LVDS0_D5_N
LVDS0_D5_P
LVDS0_D6_N
LVDS0_D6_P
LVDS0_D7_N
LVDS0_D7_P
LVDS0_CLKI0_N
LVDS0_CLKI0_P
LVDS0_CLKI1_N
LVDS0_CLKI1_P
LVDS0_CLKO0_N
LVDS0_CLKO0_P
CAM0_CLK_PLL
CAM0_TRIG_0
CAM0_TRIG_1
CAM0_TRIG_2
CAM0_MON_0
CAM0_MON_1
LVDS1_D0_N
LVDS1_D0_P
LVDS1_D1_N
LVDS1_D1_P
LVDS1_D2_N
LVDS1_D2_P
LVDS1_D3_N
LVDS1_D3_P
LVDS1_D4_N
LVDS1_D4_P
LVDS1_D5_N
LVDS1_D5_P
LVDS1_D6_N
LVDS1_D6_P
LVDS1_D7_N
LVDS1_D7_P
LVDS1_CLKI0_N
LVDS1_CLKI0_P
LVDS1_CLKI1_N
LVDS1_CLKI1_P
LVDS1_CLKO0_N
LVDS1_CLKO0_P
CAM1_CLK_PLL
CAM1_TRIG_0
CAM1_TRIG_1
CAM1_TRIG_2
CAM1_MON_0
CAM1_MON_1
CAM_RSTN
SPI0_MOSI
SPI0_MISO
SPI0_SCLK
SPI0_SS_CAM0
SPI0_SS_CAM1
SPI0_SS_ACC0
SPI0_SS_ACC1
FPGA Connector Interface - JX1
LVDS0_D0_N
LVDS0_D0_P
LVDS0_D1_N
LVDS0_D1_P
LVDS0_D2_N
LVDS0_D2_P
LVDS0_D3_N
LVDS0_D3_P
LVDS0_D4_N
LVDS0_D4_P
LVDS0_D5_N
LVDS0_D5_P
LVDS0_D6_N
LVDS0_D6_P
LVDS0_D7_N
LVDS0_D7_P
LVDS0_CLKI0_N
LVDS0_CLKI0_P
LVDS0_CLKI1_N
LVDS0_CLKI1_P
LVDS0_CLKO0_N
LVDS0_CLKO0_P
CAM0_CLK_PLL
CAM0_TRIG_0
CAM0_TRIG_1
CAM0_TRIG_2
CAM0_MON_0
CAM0_MON_1
FPGA_DONE
CARRIER_RST
JTAG_TDI
JTAG_TDO
JTAG_TCK
JTAG_TMS
PS_LED0
PS_LED1
PG_MOD
FPGA Connector Interface - JX2
LVDS1_D0_N
LVDS1_D0_P
LVDS1_D1_N
LVDS1_D1_P
LVDS1_D2_N
LVDS1_D2_P
LVDS1_D3_N
LVDS1_D3_P
LVDS1_D4_N
LVDS1_D4_P
LVDS1_D5_N
LVDS1_D5_P
LVDS1_D6_N
LVDS1_D6_P
LVDS1_D7_N
LVDS1_D7_P
LVDS1_CLKI0_N
LVDS1_CLKI0_P
LVDS1_CLKI1_N
LVDS1_CLKI1_P
LVDS1_CLKO0_N
LVDS1_CLKO0_P
CAM1_CLK_PLL
CAM1_TRIG_0
CAM1_TRIG_1
CAM1_TRIG_2
CAM1_MON_0
CAM1_MON_1
PS_LED0
PS_LED1
PG_MOD
VCCIO_EN
5 Channel Sequenced Power Supply
VCCIO_EN
PG_MOD
5V 20 Watt DC/DC Converter
Configuration and Debug
FPGA_DONE
CARRIER_RST
JTAG_TDI
JTAG_TDO
JTAG_TCK
JTAG_TMS
PS_LED0
PS_LED1
PG_MOD
I2C_SDA
I2C_SCL
RTC_INT
MAC_ID
PL_LED0
PL_LED1
PL_LED2
PL_LED3
FPGA Connector Interface - JX3
CAM_RSTN
SPI0_MOSI
SPI0_MISO
SPI0_SCLK
SPI0_SS_CAM0
SPI0_SS_CAM1
SPI0_SS_ACC0
SPI0_SS_ACC1
PL_LED3
PL_LED2
PL_LED1
PL_LED0
MAC_ID
RTC_INT
I2C1_SDA
I2C1_SCL
mSATA1_TX_N
mSATA1_TX_P
mSATA1_RX_N
mSATA1_RX_P
mSATA2_TX_N
mSATA2_TX_P
mSATA2_RX_N
mSATA2_RX_P
USB_D_N
USB_D_P
USB_PWREN
USB-VBUS
GETH_MD1_N
GETH_MD1_P
GETH_MD2_N
GETH_MD2_P
GETH_MD3_N
GETH_MD3_P
GETH_MD4_N
GETH_MD4_P
GETH_LEDY
GETH_LEDG
SD_D0
SD_D1
SD_D2
SD_D3
SD_CMD
SD_CLK
SD_CD
UART_TXD
UART_RXD
SSD mSATA Interface
I2C1_SDA
I2C1_SCL
mSATA1_TX_N
mSATA1_TX_P
mSATA1_RX_N
mSATA1_RX_P
mSATA2_TX_N
mSATA2_TX_P
mSATA2_RX_N
mSATA2_RX_P
Communications
USB_D_N
USB_D_P
USB_PWREN
USB_VBUS
GETH_MD1_N
GETH_MD1_P
GETH_MD2_N
GETH_MD2_P
GETH_MD3_N
GETH_MD3_P
GETH_MD4_N
GETH_MD4_P
GETH_LEDY
GETH_LEDG
SD_D0
SD_D1
SD_D2
SD_D3
SD_CMD
SD_CLK
SD_CD
UART_TXD
UART_RXD
HPCME
ENGINEER: John Lusher
DOCUMENT #: LES5381
REV: A
Date: 4/28/2018
Time: 6:59:42 PM
Sheet: 1 of 11
File: HPCME System Interface [LES5381].SchDoc
LUSHER ENGINEERING SERVICES
Do not disclose, use, or reproduce without written approval from Lusher Engineering Services, PLLC.

HPCME: FPGA PicoZed Interface - JX1			
ENGINEER: John Lusher		**DOCUMENT #:** LES5381	**REV:** A
Date: 4/28/2018	**Time:** 6:59:43 PM	Sheet: 2 of 11	
File: HPCME [LES5381] - FPGA - JX1.SchDoc			

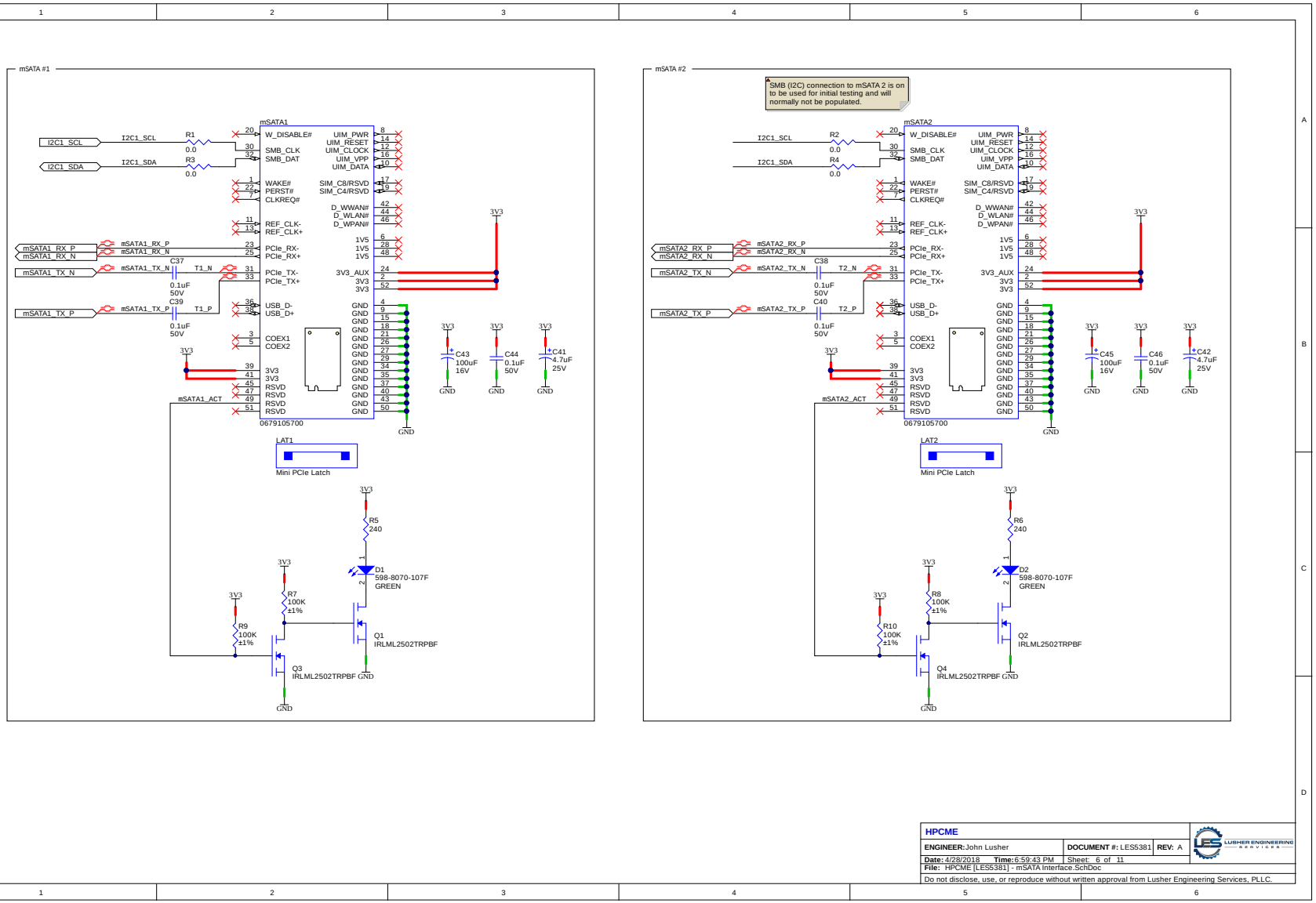


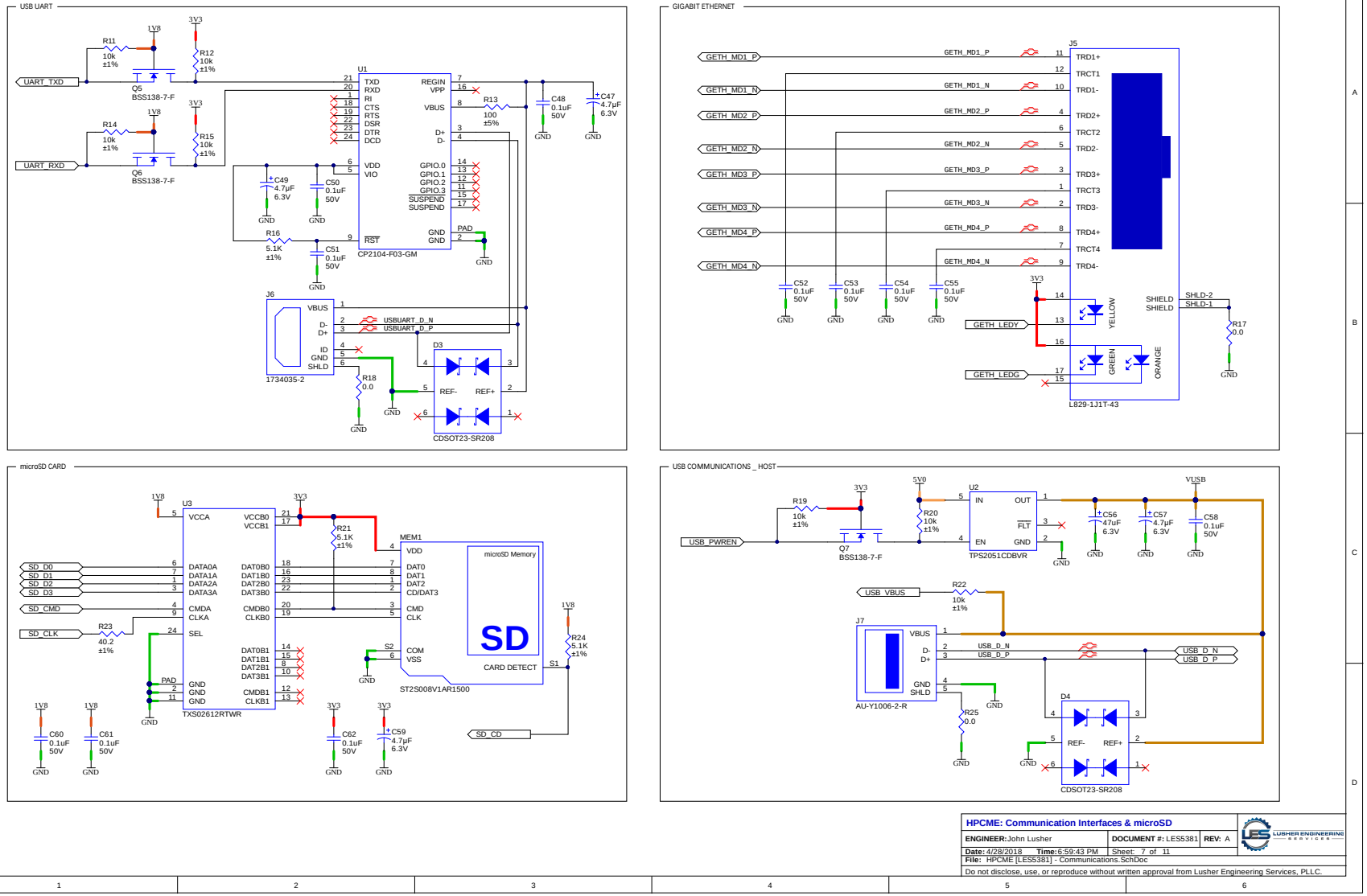
HPCME: FPGA PicoZed Interface - JX2	
ENGINEER: John Lusher	**DOCUMENT #:** LES5381 **REV:** A
Date: 4/28/2018 **Time:** 6:59:43 PM	Sheet: 3 of 11
File: HPCME [LES5381] - FPGA - JX2.SchDoc	

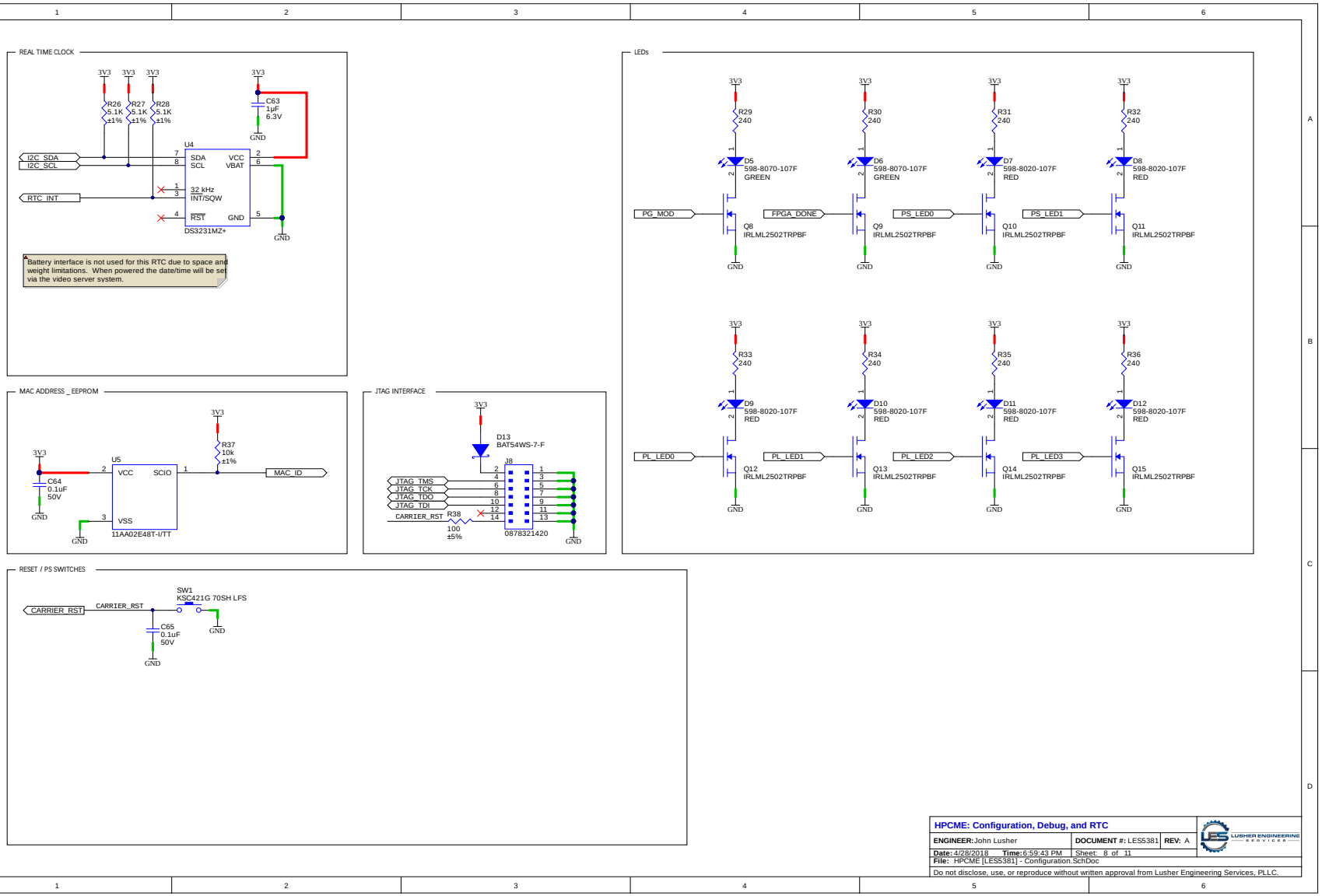

I2C0 on PS MIO pins are 1.8 VDC. I2C1 on Bank 13 are capable of 3.3 VDC.
SOM1C
AES-Z7PZ-7Z030-SOM-G
1V0
3V3
1V2
C16 100uF 4V
C14 4.7uF 25V
C15 0.1uF 50V
C20 100uF 16V
C21 4.7uF 25V
C22 0.1uF 50V
C19 100uF 4V
C17 4.7uF 25V
C18 0.1uF 50V
C24 0.1uF 50V
C23 4.7uF 25V
GND
mSATA1_TX_P
mSATA1_TX_N
mSATA2_TX_P
mSATA2_TX_N
mSATA1_RX_P
mSATA1_RX_N
mSATA2_RX_P
mSATA2_RX_N
SD_D0
SD_D2
SD_CD
SD_CLK
SD_CMD
SD_D1
SD_D3
UART_RXD
UART_TXD
GETH_LEDY
GETH_LEDG
GETH_MD1_P
GETH_MD1_N
GETH_MD2_P
GETH_MD2_N
GETH_MD3_P
GETH_MD3_N
GETH_MD4_P
GETH_MD4_N
USB_D_P
USB_D_N
USB_PWREN
USB-VBUS
SPI0_MISO
SPI0_MOSI
SPI0_SCLK
SPI0_SS_CAM0
SPI0_SS_CAM1
CAM_RSTN
SPI0_SS_ACC0
SPI0_SS_ACC1
SPI1_SCLK
I2C0_SDA
I2C0_SCL
PL_LED0
PL_LED1
PL_LED2
PL_LED3
RTC_INT
MAC_ID
SPI1_MISO
SPI1_MOSI
I2C1_SDA
I2C1_SCL
HPCME: FPGA PicoZed Interface - JX3
ENGINEER: John Lusher
DOCUMENT #: LES5381
REV: A
Date: 4/28/2018 Time: 6:59:43 PM
Sheet: 4 of 11
File: HPCME [LES5381] - FPGA - JX3.SchDoc
Do not disclose, use, or reproduce without written approval from Lusher Engineering Services, PLLC.

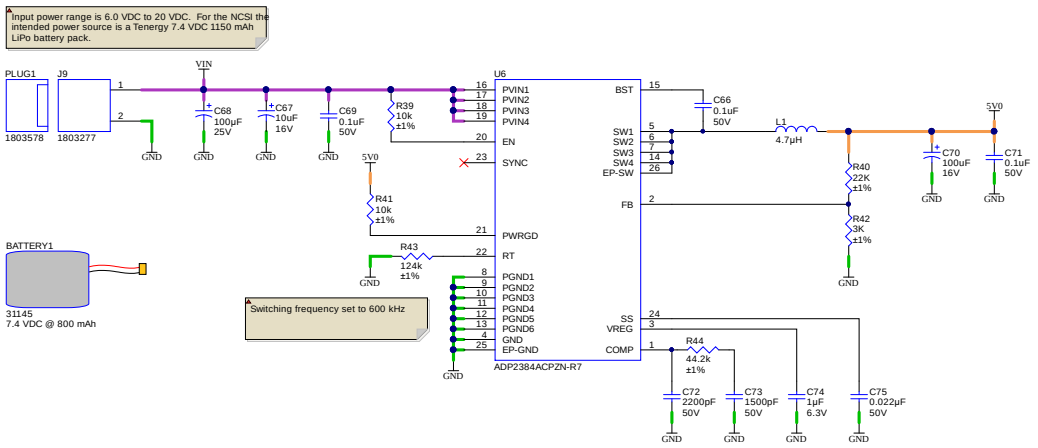


HPCME			
ENGINEER: John Lusher		**DOCUMENT #:** LES5381	**REV:** A
Date: 4/28/2018	**Time:** 6:59:43 PM	Sheet: 5 of 11	
File: HPCME [LES5381] - Sensor Interface.SchDoc			

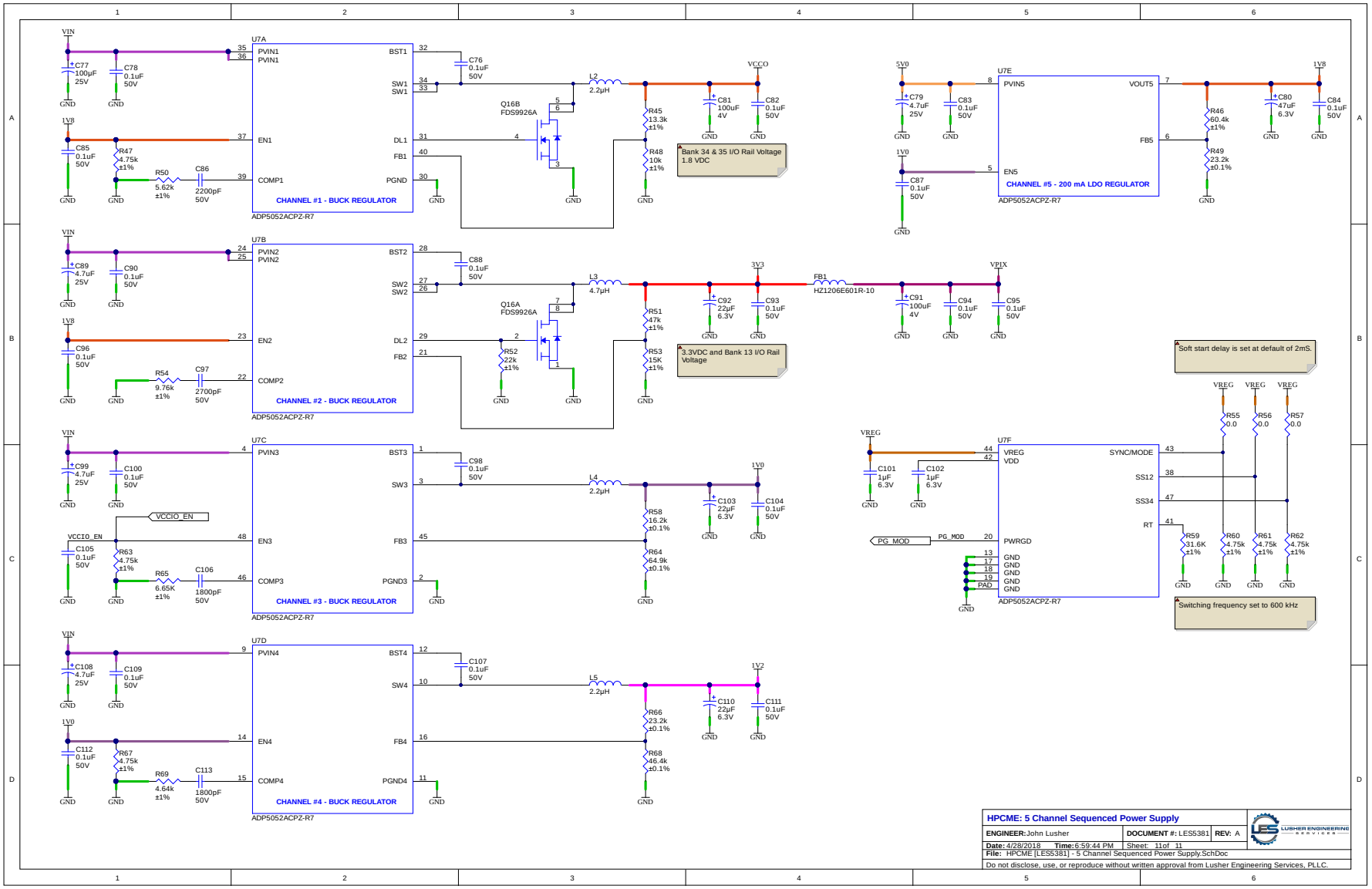

mSATA #1
mSATA1
W_DISABLE#
SMB_CLK
SMB_DAT
WAKE#
PERST#
CLKREQ#
REF_CLK-
REF_CLK+
PCIe_RX-
PCIe_RX+
PCIe_TX-
PCIe_TX+
USB_D-
USB_D+
COEX1
COEX2
3V3
RSVD
GND
UIM_PWR
UIM_RESET
UIM_CLOCK
UIM_VPP
UIM_DATA
SIM_C8/RSVD
SIM_C4/RSVD
D_WWAN#
D_WLAN#
D_WPAN#
1V5
3V3_AUX
0679105700
I2C1_SCL
I2C1_SDA
R1
R3
0.0
mSATA1_RX_P
mSATA1_RX_N
mSATA1_TX_N
mSATA1_TX_P
C37
C39
0.1uF
50V
T1_N
T1_P
mSATA1_ACT
C43
100uF
16V
C44
0.1uF
50V
C41
4.7uF
25V
LAT1
Mini PCIe Latch
R5
240
D1
598-8070-107F
GREEN
R7
100K
±1%
R9
Q1
IRLML2502TRPBF
Q3
mSATA #2
SMB (I2C) connection to mSATA 2 is on to be used for initial testing and will normally not be populated.
mSATA2
R2
R4
mSATA2_RX_P
mSATA2_RX_N
mSATA2_TX_N
mSATA2_TX_P
C38
C40
T2_N
T2_P
mSATA2_ACT
C45
100uF
16V
C46
0.1uF
50V
C42
4.7uF
25V
LAT2
R6
240
Q2
598-8070-107F
GREEN
R8
100K
±1%
R10
Q4
HPCME
ENGINEER: John Lusher
DOCUMENT #: LES5381
REV: A
Date: 4/28/2018
Time: 6:59:43 PM
Sheet: 6 of 11
File: HPCME [LES5381] - mSATA Interface.SchDoc
Do not disclose, use, or reproduce without written approval from Lusher Engineering Services, PLLC.


USB UART
GIGABIT ETHERNET
microSD CARD
USB COMMUNICATIONS _ HOST
CP2104-F03-GM
1734035-2
CDSOT23-SR208
L829-1J1T-43
TXS02612RTWR
ST2S008V1AR1500
TPS2051CDBVR
AU-Y1006-2-R
HPCME: Communication Interfaces & microSD
ENGINEER: John Lusher
DOCUMENT #: LES5381
REV: A
Date: 4/28/2018 Time: 6:59:43 PM
Sheet: 7 of 11
File: HPCME [LES5381] - Communications.SchDoc
Do not disclose, use, or reproduce without written approval from Lusher Engineering Services, PLLC.


REAL TIME CLOCK
3V3
R26 5.1K ±1%
R27 5.1K ±1%
R28 5.1K ±1%
C63 1µF 6.3V
GND
U4
I2C_SDA
I2C_SCL
SDA
SCL
VCC
VBAT
RTC_INT
32 kHz
INT/SQW
RST
GND
DS3231MZ+
Battery interface is not used for this RTC due to space and weight limitations. When powered the date/time will be set via the video server system.
LEDs
R29 240
R30 240
R31 240
R32 240
D5 598-8070-107F GREEN
D6 598-8070-107F GREEN
D7 598-8020-107F RED
D8 598-8020-107F RED
PG_MOD
FPGA_DONE
PS_LED0
PS_LED1
Q8 IRLML2502TRPBF
Q9 IRLML2502TRPBF
Q10 IRLML2502TRPBF
Q11 IRLML2502TRPBF
R33 240
R34 240
R35 240
R36 240
D9 598-8020-107F RED
D10 598-8020-107F RED
D11 598-8020-107F RED
D12 598-8020-107F RED
PL_LED0
PL_LED1
PL_LED2
PL_LED3
Q12 IRLML2502TRPBF
Q13 IRLML2502TRPBF
Q14 IRLML2502TRPBF
Q15 IRLML2502TRPBF
MAC ADDRESS _ EEPROM
U5
C64 0.1uF 50V
R37 10k ±1%
VCC
SCIO
VSS
MAC_ID
11AA02E48T-I/TT
JTAG INTERFACE
D13 BAT54WS-7-F
J8
JTAG_TMS
JTAG_TCK
JTAG_TDO
JTAG_TDI
CARRIER_RST
R38 100 ±5%
0878321420
RESET / PS SWITCHES
CARRIER_RST
SW1 KSC421G 70SH LFS
C65 0.1uF 50V
HPCME: Configuration, Debug, and RTC
ENGINEER: John Lusher
DOCUMENT #: LES5381
REV: A
Date: 4/28/2018
Time: 6:59:43 PM
Sheet: 8 of 11
File: HPCME [LES5381] - Configuration.SchDoc
LUSHER ENGINEERING SERVICES
Do not disclose, use, or reproduce without written approval from Lusher Engineering Services, PLLC.



HPCME: 5V 20W DC/DC Converter			
ENGINEER: John Lusher		**DOCUMENT #:** LES5381	**REV:** A
Date: 4/28/2018	**Time:** 6:59:44 PM	Sheet: 9 of 11	
File: HPCME [LES5381] - 5V 20W DCDC Converter.SchDoc			


U7A
PVIN1
PVIN1
BST1
SW1
SW1
EN1
DL1
FB1
COMP1
PGND
CHANNEL #1 - BUCK REGULATOR
ADP5052ACPZ-R7
Q16B
FDS9926A
L2
2.2µH
VCCO
Bank 34 & 35 I/O Rail Voltage
1.8 VDC
U7B
PVIN2
PVIN2
BST2
SW2
SW2
EN2
DL2
FB2
COMP2
CHANNEL #2 - BUCK REGULATOR
ADP5052ACPZ-R7
Q16A
FDS9926A
L3
4.7µH
3V3
3.3VDC and Bank 13 I/O Rail Voltage
FB1
HZ1206E601R-10
VPIX
U7C
PVIN3
BST3
SW3
EN3
FB3
COMP3
PGND3
CHANNEL #3 - BUCK REGULATOR
ADP5052ACPZ-R7
VCCIO_EN
L4
2.2µH
1V0
U7D
PVIN4
BST4
SW4
EN4
FB4
COMP4
PGND4
CHANNEL #4 - BUCK REGULATOR
ADP5052ACPZ-R7
L5
2.2µH
1V2
U7E
PVIN5
VOUT5
FB5
EN5
CHANNEL #5 - 200 mA LDO REGULATOR
ADP5052ACPZ-R7
5V0
U7F
VREG
VDD
SYNC/MODE
SS12
SS34
RT
PWRGD
PG_MOD
PAD
ADP5052ACPZ-R7
Soft start delay is set at default of 2mS.
Switching frequency set to 600 kHz
HPCME: 5 Channel Sequenced Power Supply
ENGINEER: John Lusher
DOCUMENT #: LES5381
REV: A
Date: 4/28/2018
Time: 6:59:44 PM
Sheet: 11of 11
File: HPCME [LES5381] - 5 Channel Sequenced Power Supply.SchDoc
Do not disclose, use, or reproduce without written approval from Lusher Engineering Services, PLLC.

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

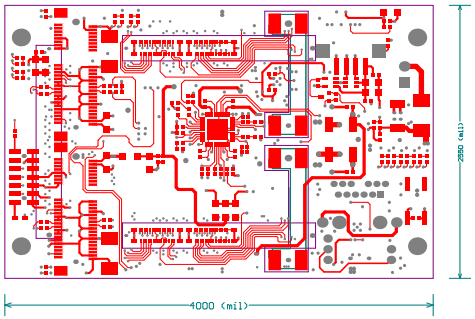
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK – 0.0 mils
TOP LAYER	TL	COPPER CLAD – 0.7 mils
		PRE-PREG – 4 mils
GROUND PLANE	PL1	COPPER CLAD – 1.4 mils
		PRE-PREG – 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD – 0.7 mils
		CORE – 15 mils
POWER PLANE	PL2	COPPER CLAD – 1.4 mils
		PRE-PREG – 5 mils
GROUND PLANE	PL3	COPPER CLAD – 1.4 mils
		CORE – 15 mils
INNER SIGNAL #2	IS2	COPPER CLAD – 0.7 mils
		PRE-PREG – 7 mils
GROUND PLANE	PL4	COPPER CLAD – 1.4 mils
		PRE-PREG – 4 mils
BOTTOM LAYER	BL	COPPER CLAD – 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK – 0.00 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: TOP LAYER		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

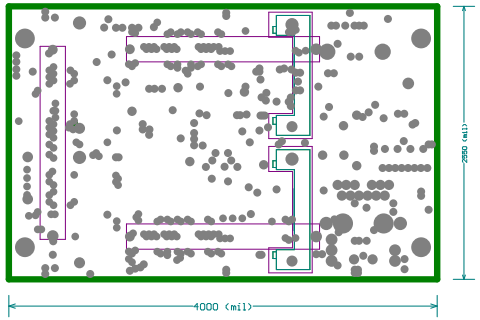
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK 0.8 mils
TOP LAYER	TL	COPPER CLAD 0.7 mils
		PRE-PREG 4 mils
GROUND PLANE	PL1	COPPER CLAD 1.4 mils
		PRE-PREG 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD 0.7 mils
		CORE 19 mils
POWER PLANE	PL2	COPPER CLAD 1.4 mils
		PRE-PREG 5 mils
GROUND PLANE	PL3	COPPER CLAD 1.4 mils
		CORE 19 mils
INNER SIGNAL #2	IS2	COPPER CLAD 0.7 mils
		PRE-PREG 7 mils
GROUND PLANE	PL4	COPPER CLAD 1.4 mils
		PRE-PREG 4 mils
BOTTOM LAYER	BL	COPPER CLAD 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK 0.80 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: GND PLANE (GND)		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

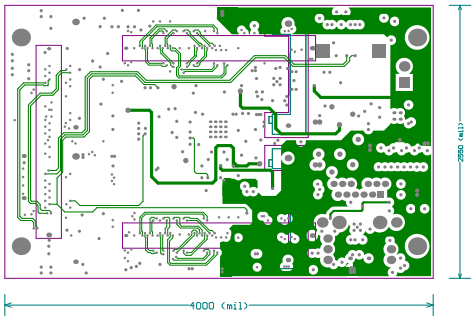
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK – 0.0 mils
TOP LAYER	TL	COPPER CLAD – 0.7 mils
		PRE-PREG – 4 mils
GROUND PLANE	PL1	COPPER CLAD – 1.4 mils
		PRE-PREG – 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD – 0.7 mils
		CORE – 15 mils
POWER PLANE	PL2	COPPER CLAD – 1.4 mils
		PRE-PREG – 5 mils
GROUND PLANE	PL3	COPPER CLAD – 1.4 mils
		CORE – 15 mils
INNER SIGNAL #2	IS2	COPPER CLAD – 0.7 mils
		PRE-PREG – 7 mils
GROUND PLANE	PL4	COPPER CLAD – 1.4 mils
		PRE-PREG – 4 mils
BOTTOM LAYER	BL	COPPER CLAD – 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK – 0.00 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: High Speed Layer #1		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

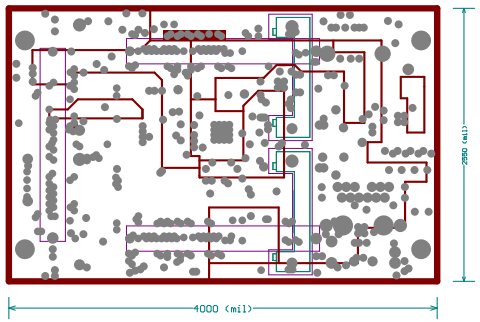
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK – 0.0 mils
TOP LAYER	TL	COPPER CLAD – 0.7 mils
		PRE-PREG – 4 mils
GROUND PLANE	PL1	COPPER CLAD – 1.4 mils
		PRE-PREG – 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD – 0.7 mils
		CORE – 15 mils
POWER PLANE	PL2	COPPER CLAD – 1.4 mils
		PRE-PREG – 5 mils
GROUND PLANE	PL3	COPPER CLAD – 1.4 mils
		CORE – 15 mils
INNER SIGNAL #2	IS2	COPPER CLAD – 0.7 mils
		PRE-PREG – 7 mils
GROUND PLANE	PL4	COPPER CLAD – 1.4 mils
		PRE-PREG – 4 mils
BOTTOM LAYER	BL	COPPER CLAD – 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK – 0.00 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: POWER PLANE ((Multiple Nets))		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

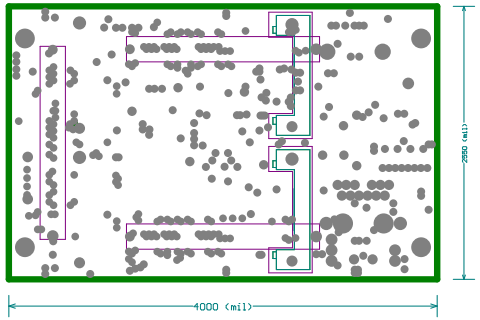
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK 0.8 mils
TOP LAYER	TL	COPPER CLAD 0.7 mils
		PRE-PREG 4 mils
GROUND PLANE	PL1	COPPER CLAD 1.4 mils
		PRE-PREG 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD 0.7 mils
POWER PLANE	PL2	CORE 19 mils
		COPPER CLAD 1.4 mils
		PRE-PREG 5 mils
GROUND PLANE	PL3	COPPER CLAD 1.4 mils
		CORE 19 mils
INNER SIGNAL #2	IS2	COPPER CLAD 0.7 mils
		PRE-PREG 7 mils
GROUND PLANE	PL4	COPPER CLAD 1.4 mils
		PRE-PREG 4 mils
BOTTOM LAYER	BL	COPPER CLAD 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK 0.80 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: GND PLANE (GND)		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

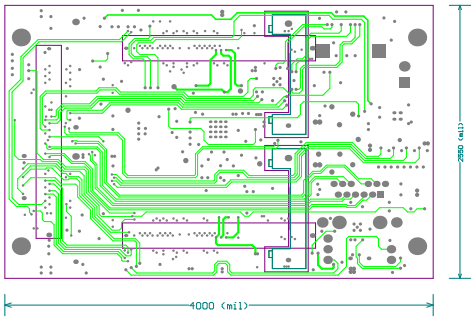
PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
BOARD SIZE: 4.000 in. X 2.550 in.
LAYERS: 8
COPPER THICKNESS:
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
OUTER: 17 um (0.5 oz)
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
INNER SIGNAL: 17 um (0.5 oz)
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
INNER POWER: 35 um (1 oz)
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
VIAS: FEED-THROUGH TENTED
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK – 0.5 mils
TOP LAYER	TL	COPPER CLAD – 0.7 mils
		PRE-PREG – 4 mils
GROUND PLANE	PL1	COPPER CLAD – 1.4 mils
		PRE-PREG – 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD – 0.7 mils
		CORE – 15 mils
POWER PLANE	PL2	COPPER CLAD – 1.4 mils
		PRE-PREG – 5 mils
GROUND PLANE	PL3	COPPER CLAD – 1.4 mils
		CORE – 15 mils
INNER SIGNAL #2	IS2	COPPER CLAD – 0.7 mils
		PRE-PREG – 7 mils
GROUND PLANE	PL4	COPPER CLAD – 1.4 mils
		PRE-PREG – 4 mils
BOTTOM LAYER	BL	COPPER CLAD – 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK – 0.50 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808	SCALE: SCALE: 1.00
		DRAWN	JOHN LUSHER II	9.MAY.16		DATE: 4/28/2018
		APPROVED	JOHN LUSHER II	9.MAY.16		
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:	SHEET 1 of 1 B
					LES5381 – HPCME NDE SYSTEM	
TEMPLATE: High Speed Layer #2		PCB: – HPCME System Interface [LES5381]PCBDoc				

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

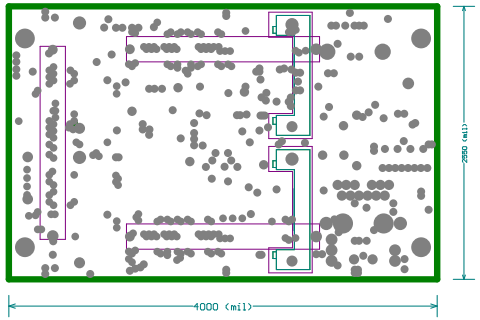
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK 0.8 mils
TOP LAYER	TL	COPPER CLAD 0.7 mils
		PRE-PREG 4 mils
GROUND PLANE	PL1	COPPER CLAD 1.4 mils
		PRE-PREG 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD 0.7 mils
		CORE 19 mils
POWER PLANE	PL2	COPPER CLAD 1.4 mils
		PRE-PREG 5 mils
GROUND PLANE	PL3	COPPER CLAD 1.4 mils
		CORE 19 mils
INNER SIGNAL #2	IS2	COPPER CLAD 0.7 mils
		PRE-PREG 7 mils
GROUND PLANE	PL4	COPPER CLAD 1.4 mils
		PRE-PREG 4 mils
BOTTOM LAYER	BL	COPPER CLAD 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK 0.80 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title: LES5381 – HPCME NDE SYSTEM
TEMPLATE: GND PLANE (GND)		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1260
VIA'S	507
SLOTS	0
HOLES	574

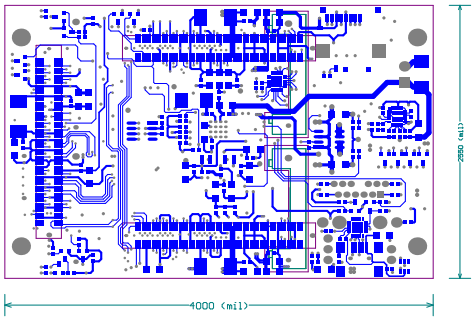
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK – 0.0 mils
TOP LAYER	TL	COPPER CLAD – 0.7 mils
		PRE-PREG – 4 mils
GROUND PLANE	PL1	COPPER CLAD – 1.4 mils
		PRE-PREG – 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD – 0.7 mils
		CORE – 15 mils
POWER PLANE	PL2	COPPER CLAD – 1.4 mils
		PRE-PREG – 5 mils
GROUND PLANE	PL3	COPPER CLAD – 1.4 mils
		CORE – 15 mils
INNER SIGNAL #2	IS2	COPPER CLAD – 0.7 mils
		PRE-PREG – 7 mils
GROUND PLANE	PL4	COPPER CLAD – 1.4 mils
		PRE-PREG – 4 mils
BOTTOM LAYER	BL	COPPER CLAD – 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK – 0.00 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: BOTTOM LAYER		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

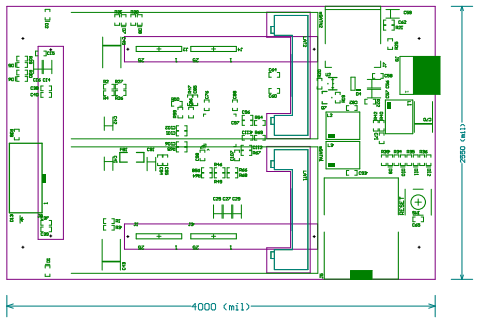
BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
LAYERS: 8
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

BOARD SIZE: 4.000 in. X 2.550 in.
COPPER THICKNESS:
OUTER: 17 um (0.5 oz)
INNER SIGNAL: 17 um (0.5 oz)
INNER POWER: 35 um (1 oz)
VIAS: FEED-THROUGH TENTED

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK – 0.5 mils
TOP LAYER	TL	COPPER CLAD – 0.7 mils
		PRE-PREG – 4 mils
GROUND PLANE	PL1	COPPER CLAD – 1.4 mils
		PRE-PREG – 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD – 0.7 mils
		CORE – 15 mils
POWER PLANE	PL2	COPPER CLAD – 1.4 mils
		PRE-PREG – 5 mils
GROUND PLANE	PL3	COPPER CLAD – 1.4 mils
		CORE – 15 mils
INNER SIGNAL #2	IS2	COPPER CLAD – 0.7 mils
		PRE-PREG – 7 mils
GROUND PLANE	PL4	COPPER CLAD – 1.4 mils
		PRE-PREG – 4 mils
BOTTOM LAYER	BL	COPPER CLAD – 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK – 0.50 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: Top Silkscreen Overlay		PCB: – HPCME System Interface [LES5381].PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

BUILD STATISTICS

GENERAL

NETS	276
COMPONENTS	240

PRIMITIVES

FILLS	56
TRACKS	4996
STRINGS	247
ARC'S	13

LANDS / HOLES / SLOTS

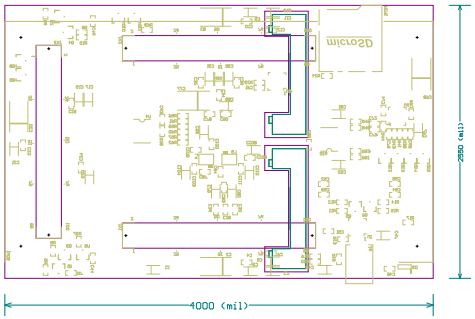
PADS	1290
VIA'S	507
SLOTS	0
HOLES	574

BUILD CONFIGURATION AND SPECIFICATIONS SUMMARY

BOARD TECHNOLOGY: 5 TRACK / 20 VIA / 5 SPACE
BOARD SIZE: 4.000 in. X 2.550 in.
LAYERS: 8
COPPER THICKNESS:
MATERIAL: FR4 (Tg 170) THICKNESS: 0.062"
OUTER: 17 um (0.5 oz)
TOLERANCE: IPC-6012 TYPE 3 CLASS 2
INNER SIGNAL: 17 um (0.5 oz)
BOW & TWIST: IPC-6012 TYPE 3 CLASS 2
INNER POWER: 35 um (1 oz)
SILKSCREEN: COMPONENT & SOLDER SIDE (WHITE)
SOLDER RESIST: SCREEN PRINT (MATTE BLACK)
VIAS: FEED-THROUGH TENTED
TRACK FINISH: IMMERSION GOLD
PROFILING: CUT AND TRIM TO MECHANICAL LAYER #1
N.C. ROUTE

ACCEPTABILITY: MATERIALS AND WORKMANSHIP FOR ALL PRINTED WIRING BOARDS TO MEET OR EXCEED THE REQUIREMENTS OF:
IPC-A-600F (CLASS 2)
UL CERTIFICATION

NOTE: (MIN. TRACK / PADS / CLEARANCE IN 0.001")



LAYER STACK CONFIGURATION

8 LAYER BUILD

TOP SOLDER MASK	TS	SOLDER MASK – 0.8 mils
TOP LAYER	TL	COPPER CLAD – 0.7 mils
		PRE-PREG – 4 mils
GROUND PLANE	PL1	COPPER CLAD – 1.4 mils
		PRE-PREG – 7 mils
INNER SIGNAL #1	IS1	COPPER CLAD – 0.7 mils
		CORE – 15 mils
POWER PLANE	PL2	COPPER CLAD – 1.4 mils
		PRE-PREG – 5 mils
GROUND PLANE	PL3	COPPER CLAD – 1.4 mils
		CORE – 15 mils
INNER SIGNAL #2	IS2	COPPER CLAD – 0.7 mils
		PRE-PREG – 7 mils
GROUND PLANE	PL4	COPPER CLAD – 1.4 mils
		PRE-PREG – 4 mils
BOTTOM LAYER	BL	COPPER CLAD – 0.7 mils
BOTTOM SOLDER MASK	BS	SOLDER MASK – 0.80 mils

TOTAL BOARD THICKNESS: (APPROX)
0.066 " +/- 0.005"

DIMENSIONS IN INCHES

		PRODUCT	JT5381 – NCSI	DATE	LUSHER ENGINEERING SERVICES 8754 Tejas Ranch Loop Bryan, TX 77808
		DRAWN	JOHN LUSHER II	9.MAY.16	
		APPROVED	JOHN LUSHER II	9.MAY.16	
A	RELEASE VERSION	CUSTOMER	Lusher Engineering Services		Title:
					LES5381 – HPCME NDE SYSTEM
TEMPLATE: Bottom Silkscreen Overlay		PCB: – HPCME System Interface [LES5381]PCBDoc			

SCALE: SCALE: 1.00
DATE: 4/28/2018
SHEET 1 of 1 B

APPENDIX C

NODE DEGREE ENGINE - FPGA DESIGN

```
----------------------------------------------------------------------------------------------------------------------------------
--  FILE:               cca_core.vhd
--  ENTITY:             cca_core
--
--  Company:            Texas A&M University
--  Author:             John Lusher II, P.E.
--  Project Name:       fMRI Brain Connectivity Maps Processing Engine
--  Module Name:        fMRI -  RTL
--  Create Date:        January 4, 2017
--  Target Devices:     Xilinx UltraScale XCKU040 Family / KCU105 Development Board
--
--  Core Description:   CCA Engine
--
--  Revisions:
--  1.0: File Created
--  1.1: Updated for PoC System - 64 core  (Aug 2017)
--  1.2: Updated for HPDME System - 32 core (Nov 2017)
----------------------------------------------------------------------------------------------------------------------------------

-- Standard IEEE Library
library IEEE;
    -- Logic Elements
    use IEEE.STD_LOGIC_1164.ALL;
    -- Unsigned Logic Elements
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
    -- Numeric
    use IEEE.NUMERIC_STD.ALL;

--! Xilinx Virtex Components Library
library UNISIM;
    --! Xilinx Virtex Components
    use UNISIM.Vcomponents.ALL;

entity cca_core is
    Generic
    (
        DATA_WIDTH_BYTES                : natural := 4;
        DATA_WIDTH_BITS                 : natural := 32;
        NUM_R_CORES                     : natural := 8;
        NUM_SEEDS                       : natural := 16;
        NUM_VOXELS                      : natural := 2
    );
    Port
    (
                                                                        ----------------------------------------------------
        aclk                            : IN   STD_LOGIC;               --! AXI Clock                          (Active High)
        axi_aresetn                     : IN   STD_LOGIC;               --! AXI Reset                           (Active Low)
        system_reset                    : IN   STD_LOGIC;               --! System Reset                       (Active High)
                                                                        ----------------------------------------------------

        N_SAMPLES                       : IN   STD_LOGIC_VECTOR ( 11 downto 0 );
        R_PRIME                         : IN   STD_LOGIC_VECTOR ( 31 downto 0 );
        CURRENT_ADDR                    : OUT  STD_LOGIC_VECTOR ( 16 downto 0 );


        clk                             : OUT  STD_LOGIC;
        RESULTADDR                      : OUT  STD_LOGIC_VECTOR ( 12 downto 0 );
        VOXELCOUNT                      : OUT  STD_LOGIC_VECTOR ( 11 downto 0 );
        VOXELCOUNTACC                   : OUT  STD_LOGIC_VECTOR ( 11 downto 0 );
        WR                              : OUT  STD_LOGIC_VECTOR ( 0 downto 0 );
```

```
        RESULT0                         : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        RESULTVO                        : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        RESULTSD                        : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        VOXIDX                          : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        BRAMP0S0                        : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        SEED                            : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        VOXEL                           : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        MULRESULT                       : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        MULVALID                        : OUT  STD_LOGIC;
        LASTIN                          : OUT  STD_LOGIC;
        LASTOUT0                        : OUT  STD_LOGIC;
        LASTOUT1                        : OUT  STD_LOGIC;
        LASTOUT2                        : OUT  STD_LOGIC;
        LASTOUT3                        : OUT  STD_LOGIC;
        LASTOUT4                        : OUT  STD_LOGIC;
        A                               : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
        B                               : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );
                                                                --------------------------------------------------
                                                                --  Voxel AXI Stream Interface
        M_AXIS_MM2S_VOXELS_tdata        : IN   STD_LOGIC_VECTOR((DATA_WIDTH_BITS * NUM_VOXELS) - 1 downto 0);
        M_AXIS_MM2S_VOXELS_tkeep        : IN   STD_LOGIC_VECTOR((DATA_WIDTH_BYTES * NUM_VOXELS) - 1 downto 0);
        M_AXIS_MM2S_VOXELS_tlast        : IN   STD_LOGIC;                       --! Data Last                          (Active High)
        M_AXIS_MM2S_VOXELS_tready       : OUT  STD_LOGIC;                       --! Interface Ready                    (Active High)
        M_AXIS_MM2S_VOXELS_tvalid       : IN   STD_LOGIC;                       --! Data Valid                         (Active High)
                                                                --------------------------------------------------

                                                                --------------------------------------------------
        SEEDS_BRAM_PORT_0_addr          : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );-- Seed BRAM Port
        SEEDS_BRAM_PORT_0_clk           : OUT  STD_LOGIC;                       --
        SEEDS_BRAM_PORT_0_din           : OUT  STD_LOGIC_VECTOR ( 511 downto 0 );
        SEEDS_BRAM_PORT_0_dout          : IN   STD_LOGIC_VECTOR ( 511 downto 0 );
        SEEDS_BRAM_PORT_0_en            : OUT  STD_LOGIC;                       --
        SEEDS_BRAM_PORT_0_rst           : OUT  STD_LOGIC;                       --
        SEEDS_BRAM_PORT_0_we            : OUT  STD_LOGIC_VECTOR ( 63 downto 0 );--
                                                                --------------------------------------------------

                                                                --------------------------------------------------
        RESULT_BRAM_PORT_0_addr         : OUT  STD_LOGIC_VECTOR ( 31 downto 0 );-- Result BRAM Port
        RESULT_BRAM_PORT_0_clk          : OUT  STD_LOGIC;                       --
        RESULT_BRAM_PORT_0_din          : OUT  STD_LOGIC_VECTOR ( 63 downto 0 );--
        RESULT_BRAM_PORT_0_en           : OUT  STD_LOGIC;                       --
        RESULT_BRAM_PORT_0_rst          : OUT  STD_LOGIC;                       --
        RESULT_BRAM_PORT_0_we           : OUT  STD_LOGIC_VECTOR ( 7 downto 0 )  --
                                                                --------------------------------------------------
    );
end cca_core;

------------------------------------------------------------------------------------------------------------------------
--  Architecture:   cca_core (RTL)
------------------------------------------------------------------------------------------------------------------------
--! @brief      Architecture definition of entity: cca_core
--! @details    The logic and I/O interface for the fMRI CCA Engine
architecture RTL of cca_core is
    ------------------------------------------------------------------------------------------------------------------------
    -- Attributes
    ------------------------------------------------------------------------------------------------------------------------

    ------------------------------------------------------------------------------------------------------------------------
    -- Signals
```

```
    ------------------------------------------------------------------------------------------------------------------------
                                                                                --! Array to hold 128 seeds (32-bit single precision data)
    type VXIDX_ARRAY is array(0 to (NUM_VOXELS)) of STD_LOGIC_VECTOR(31 downto 0);
    type DATA_ARRAY is array(0 to (NUM_SEEDS * NUM_VOXELS)) of STD_LOGIC_VECTOR(31 downto 0);
    type LOGIC_ARRAY is array(0 to (NUM_SEEDS * NUM_VOXELS)) of STD_LOGIC;      --! Logic array
    type COUNT_ARRAY is array(0 to (NUM_SEEDS * NUM_VOXELS)) of STD_LOGIC_VECTOR(11 downto 0);
    type LAST_COUNT_ARRAY is array(0 to (NUM_SEEDS * NUM_VOXELS)) of STD_LOGIC_VECTOR(3 downto 0);

    ------------------------------------------------------------------------------------------------------------------------
    -- Covar Signals
    ------------------------------------------------------------------------------------------------------------------------
    signal M_AXIS_MM2S_VOXELS_ready     : LOGIC_ARRAY;                          --! Voxel Input Ready
    signal M_AXIS_MM2S_SEEDS_ready      : LOGIC_ARRAY;                          --! Seed Input Ready

    signal multiplier_result            : DATA_ARRAY;                           --! Multiplier Result
    signal multiplier_valid             : LOGIC_ARRAY;                          --! Multiplier Valid
    signal multiplier_last              : LOGIC_ARRAY;                          --! Multiplier Last

    signal got_data                     : LOGIC_ARRAY;                          --! Got Data Flag
    signal accumulator_valid            : LOGIC_ARRAY;                          --! Accumulator Valid
    signal accumulator_last_in          : LOGIC_ARRAY;                          --! Accumulator Last Input
    signal accumulator_last_out         : LOGIC_ARRAY;                          --! Accumulator Last Input
    signal accumulator_ready            : LOGIC_ARRAY;                          --! Accumulator Ready
    signal voxel_count                  : COUNT_ARRAY;                          --! Voxel Counter
    signal voxel_count_acc              : COUNT_ARRAY;                          --! Voxel Counter - Accumulator
    signal result_address               : STD_LOGIC_VECTOR(16 downto 0);        --! Result Address
    signal sample_count_1               : COUNT_ARRAY;                          --! N Samples + 1 (SD)
    signal sample_count_2               : COUNT_ARRAY;                          --! N Samples + 2 (Voxel #)
    signal sample_count_3               : COUNT_ARRAY;                          --! N Samples - 7 (Voxel #)

    signal M_AXIS_MM2S_VOXELS_tready_T  : STD_LOGIC_VECTOR(((NUM_VOXELS * NUM_SEEDS)-1) downto 0);

    signal RESULTS_0                    : DATA_ARRAY;                           --! Holding Variable Array
    signal RESULTS_1                    : DATA_ARRAY;                           --! Holding Variable Array
    signal RESULTS_COV                  : DATA_ARRAY;                           --! Holding Variable Array
    signal RESULTS_SD                   : DATA_ARRAY;                           --! Holding Variable Array
    signal VOXELINDEX                   : VXIDX_ARRAY;                          --! Voxel # index
    signal last_count                   : LAST_COUNT_ARRAY;                     --! Voxel / SD State Counter


    signal accumulator_last_out_1       : LOGIC_ARRAY;                          --! Accumulator Last Input
    signal accumulator_last_out_2       : LOGIC_ARRAY;                          --! Accumulator Last Input
    signal accumulator_last_out_3       : LOGIC_ARRAY;                          --! Accumulator Last Input
    signal accumulator_last_out_4       : LOGIC_ARRAY;                          --! Accumulator Last Input

    signal accumulator_reset            : LOGIC_ARRAY;                          --! Accumulator Reset

    signal seed_count                   : STD_LOGIC_VECTOR(11 downto 0);        --! Seed Address Counter



    ------------------------------------------------------------------------------------------------------------------------
    -- Divider Signals ("r" core)
    ------------------------------------------------------------------------------------------------------------------------
    type R_DATA_ARRAY is array(0 to (NUM_R_CORES - 1)) of STD_LOGIC_VECTOR(31 downto 0);
    type R_INDEX_ARRAY is array(0 to (NUM_R_CORES - 1)) of STD_LOGIC_VECTOR(26 downto 0);
    type R_INDEXGRP_ARRAY is array(0 to (NUM_R_CORES - 1)) of STD_LOGIC_VECTOR(1 downto 0);
    type R_INDEXLOW_ARRAY is array(0 to (NUM_R_CORES - 1)) of STD_LOGIC_VECTOR(2 downto 0);
    type R_COMP_ARRAY is array(0 to (NUM_R_CORES - 1)) of STD_LOGIC_VECTOR(7 downto 0);
    type R_COUNT_ARRAY is array(0 to (NUM_R_CORES - 1)) of STD_LOGIC_VECTOR(7 downto 0);
```

```
    type R_LOGIC_ARRAY is array(0 to (NUM_R_CORES - 1)) of STD_LOGIC;          --! Logic array

    signal a_dividend_data              : R_DATA_ARRAY;                        --! Dividend data
    signal a_dividend_valid             : R_LOGIC_ARRAY;                       --! Dividend valid
    signal b_divisor_data               : R_DATA_ARRAY;                        --! Divisor data
    signal b_divisor_valid              : R_LOGIC_ARRAY;                       --! Divisor valid
    signal r_quotient_data              : R_DATA_ARRAY;                        --! Quotient data
    signal r_quotient_valid             : R_LOGIC_ARRAY;                       --! Quotient valid

    ------------------------------------------------------------------------------------------------------------------------
    -- Comparator Signals ("r" core)
    ------------------------------------------------------------------------------------------------------------------------
    signal b_comp_data                  : STD_LOGIC_VECTOR(31 DOWNTO 0);       --! "B" data
    signal b_comp_valid                 : STD_LOGIC;                           --! "B" valid
    signal r_comp_data                  : R_COMP_ARRAY;                        --! Result data GTE
    signal r_comp_valid                 : R_LOGIC_ARRAY;                       --! Result valid GTE
    signal r_complte_data               : R_COMP_ARRAY;                        --! Result data LTE
    signal r_complte_valid              : R_LOGIC_ARRAY;                       --! Result valid LTE
    signal r_value                      : R_DATA_ARRAY;                        --! "r" value
    signal r_capture                    : R_LOGIC_ARRAY;                       --! "r" capture flag (1 = save data)
    signal post_comp_valid              : R_COUNT_ARRAY;                       --! Post valid count array
    signal r_value_low_index            : R_INDEXLOW_ARRAY;                    --! "r" index (subgroup) value (lower 3 bits)
    signal r_value_grp_index            : R_INDEXGRP_ARRAY;                    --! "r" index (r-group) value (next 2 bits)
    signal r_value_index                : R_INDEX_ARRAY;                       --! "r" index value (voxel #)


    ------------------------------------------------------------------------------------------------------------------------
    --  Voxel AXI Stream Interface - Delay Register
    ------------------------------------------------------------------------------------------------------------------------
    signal REG_VOXELS_tdata             : STD_LOGIC_VECTOR((DATA_WIDTH_BITS * NUM_VOXELS) - 1 downto 0);
    signal REG_VOXELS_tlast             : STD_LOGIC;                           --! Data Last                          (Active High)
    signal REG_VOXELS_tvalid            : STD_LOGIC;                           --! Data Valid                         (Active High)
                                                                               ------------------------------------------------------


    ------------------------------------------------------------------------------------------------------------------------
    -- Components
    ------------------------------------------------------------------------------------------------------------------------
    ------------------------------------------------------------------------------------------------------------------------
    -- Floating Point Multiplier Component
    -- Result = A * B
    ------------------------------------------------------------------------------------------------------------------------
    component floating_point_mul is
    port
    (
        aclk                            : IN   STD_LOGIC;                      --! Clock                              (Active High)
        s_axis_a_tvalid                 : IN   STD_LOGIC;                      --! Port "A" Data Valid                (Active High)
        s_axis_a_tready                 : OUT  STD_LOGIC;                      --! Port "A" Data Ready                (Active High)
        s_axis_a_tdata                  : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);  --! Port "A" Data                        (32-bit Bus)
        s_axis_a_tlast                  : IN   STD_LOGIC;                      --! Port "A" Data Last                 (Active High)
        s_axis_b_tvalid                 : IN   STD_LOGIC;                      --! Port "B" Data Valid                (Active High)
        s_axis_b_tready                 : OUT  STD_LOGIC;                      --! Port "B" Data Ready                (Active High)
        s_axis_b_tdata                  : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);  --! Port "B" Data                        (32-bit Bus)
        s_axis_b_tlast                  : IN   STD_LOGIC;                      --! Port "B" Data Last                 (32-bit Bus)
        m_axis_result_tvalid            : OUT  STD_LOGIC;                      --! Result Valid                       (Active High)
        m_axis_result_tready            : IN   STD_LOGIC;                      --! Result Ready                       (Active High)
        m_axis_result_tdata             : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);  --! Result Data                        (32-bit Bus)
        m_axis_result_tlast             : OUT  STD_LOGIC                       --! Result Last                       (Active High)
    );
    end component;
```

```
-------------------------------------------------------------------------------------------------------------------------------
-- Floating Point Accumulator Component
-- Result = Result + A
-------------------------------------------------------------------------------------------------------------------------------
component floating_point_acc is
port
(
    aclk                        : IN  STD_LOGIC;                        --! Clock                          (Active High)
    aresetn                     : IN  STD_LOGIC;                        --! AReset                          (Active Low)
    s_axis_a_tvalid             : IN  STD_LOGIC;                        --! Port "A" Data Valid             (Active High)
    s_axis_a_tready             : OUT STD_LOGIC;                        --! Port "A" Data Ready             (Active High)
    s_axis_a_tdata              : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);    --! Port "A" Data                    (32-bit Bus)
    s_axis_a_tlast              : IN  STD_LOGIC;                        --! Port "A" Last                   (Active High)
    m_axis_result_tvalid        : OUT STD_LOGIC;                        --! Result Valid                    (Active High)
    m_axis_result_tready        : IN  STD_LOGIC;                        --! Result Ready                    (Active High)
    m_axis_result_tdata         : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);    --! Result Data                    (32-bit Bus)
    m_axis_result_tlast         : OUT STD_LOGIC                         --! Result Last                   (Active High)
);
end component;

-------------------------------------------------------------------------------------------------------------------------------
-- Floating Point Divider Component
-- Result = A / B
-------------------------------------------------------------------------------------------------------------------------------
component floating_point_div is
port
(
    aclk                        : IN   STD_LOGIC;                       --! Clock                          (Active High)
    s_axis_a_tvalid             : IN   STD_LOGIC;                       --! Port "A" Data Valid             (Active High)
    s_axis_a_tready             : OUT  STD_LOGIC;                       --! Port "A" Data Ready             (Active High)
    s_axis_a_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "A" Data                    (32-bit Bus)
    s_axis_b_tvalid             : IN   STD_LOGIC;                       --! Port "B" Data Valid             (Active High)
    s_axis_b_tready             : OUT  STD_LOGIC;                       --! Port "B" Data Ready             (Active High)
    s_axis_b_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "B" Data                    (32-bit Bus)
    m_axis_result_tvalid        : OUT  STD_LOGIC;                       --! Result Valid                    (Active High)
    m_axis_result_tready        : IN   STD_LOGIC;                       --! Result Ready                    (Active High)
    m_axis_result_tdata         : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0)    --! Result Data                    (32-bit Bus)
);
end component;

-------------------------------------------------------------------------------------------------------------------------------
-- Floating Point Compare Component
-- Result = A >= B
-------------------------------------------------------------------------------------------------------------------------------
component floating_point_comp is
port
(
    aclk                        : IN   STD_LOGIC;                       --! Clock                          (Active High)
    s_axis_a_tvalid             : IN   STD_LOGIC;                       --! Port "A" Data Valid             (Active High)
    s_axis_a_tready             : OUT  STD_LOGIC;                       --! Port "A" Data Ready             (Active High)
    s_axis_a_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "A" Data                    (32-bit Bus)
    s_axis_b_tvalid             : IN   STD_LOGIC;                       --! Port "B" Data Valid             (Active High)
    s_axis_b_tready             : OUT  STD_LOGIC;                       --! Port "B" Data Ready             (Active High)
    s_axis_b_tdata              : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "B" Data                    (32-bit Bus)
    m_axis_result_tvalid        : OUT  STD_LOGIC;                       --! Result Valid                    (Active High)
    m_axis_result_tready        : IN   STD_LOGIC;                       --! Result Ready                    (Active High)
    m_axis_result_tdata         : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)     --! Result Data                    (32-bit Bus)
);
```

```vhdl
    end component;

        -------------------------------------------------------------------------------------------------------------------------------
    -- Floating Point Compare Component
    -- Result = A <= B
    -------------------------------------------------------------------------------------------------------------------------------
    component floating_point_complte is
    port
    (
        aclk                                : IN   STD_LOGIC;                       --! Clock                             (Active High)
        s_axis_a_tvalid                     : IN   STD_LOGIC;                       --! Port "A" Data Valid               (Active High)
        s_axis_a_tready                     : OUT  STD_LOGIC;                       --! Port "A" Data Ready               (Active High)
        s_axis_a_tdata                      : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "A" Data                      (32-bit Bus)
        s_axis_b_tvalid                     : IN   STD_LOGIC;                       --! Port "B" Data Valid               (Active High)
        s_axis_b_tready                     : OUT  STD_LOGIC;                       --! Port "B" Data Ready               (Active High)
        s_axis_b_tdata                      : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);   --! Port "B" Data                      (32-bit Bus)
        m_axis_result_tvalid                : OUT  STD_LOGIC;                       --! Result Valid                      (Active High)
        m_axis_result_tready                : IN   STD_LOGIC;                       --! Result Ready                      (Active High)
        m_axis_result_tdata                 : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)     --! Result Data                        (32-bit Bus)
    );
    end component;


begin
    -- Set Seed BRAM Constants
    SEEDS_BRAM_PORT_0_clk   <= aclk;
    SEEDS_BRAM_PORT_0_en    <= '1';
    SEEDS_BRAM_PORT_0_we    <= X"0000000000000000";
    SEEDS_BRAM_PORT_0_rst   <= '0';
    SEEDS_BRAM_PORT_0_din   <= X"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";

    -- Set Result BRAM Constants
    RESULT_BRAM_PORT_0_clk  <= aclk;
    RESULT_BRAM_PORT_0_en   <= '1';
    RESULT_BRAM_PORT_0_rst  <= '0';

    -- Set comparator
    b_comp_data             <= R_PRIME;
    b_comp_valid            <= '1';


    -- ILA (Logic Analyzer)
    clk             <= aclk;
    RESULTADDR      <= result_address(12 downto 0);
    CURRENT_ADDR    <= result_address;
    VOXELCOUNT      <= voxel_count(0);
    VOXELCOUNTACC   <= voxel_count_acc(0);
    MULVALID        <= multiplier_valid(6);

    -- Static values of r-index low
    r_value_low_index(0)              <= "000";
    r_value_low_index(1)              <= "001";
    r_value_low_index(2)              <= "010";
    r_value_low_index(3)              <= "011";
    r_value_low_index(4)              <= "100";
    r_value_low_index(5)              <= "101";
    r_value_low_index(6)              <= "110";
    r_value_low_index(7)              <= "111";
```

```
    -- Capture results - Result Bank Voxel #0 or Voxel #1
    RESULT0     <= RESULTS_0(6);
    RESULTVO    <= RESULTS_COV(6);
    RESULTSD    <= RESULTS_SD(6);
    VOXIDX      <= VOXELINDEX(0);
    MULRESULT   <= multiplier_result(6);
    SEED        <= SEEDS_BRAM_PORT_0_dout(((3 * DATA_WIDTH_BITS) + 31)  downto (3 * DATA_WIDTH_BITS));
    VOXEL       <= REG_VOXELS_tdata(31 downto 0);
    LASTIN      <= accumulator_last_in(8);
    LASTOUT0    <= accumulator_last_out(6);
    LASTOUT1    <= accumulator_last_out_1(6);
    LASTOUT2    <= accumulator_last_out_2(6);
    LASTOUT3    <= accumulator_last_out_3(6);
    LASTOUT4    <= accumulator_last_out_4(6);


    -- Set divider A/B
    A <= a_dividend_data(6);
    B <= b_divisor_data(6);

    ------------------------------------------------------------------------------------------------------------------------
    -- Register Delay for Voxel Stream
    ------------------------------------------------------------------------------------------------------------------------
    ------------------------------------------------------------------------------------------------------------------------
    -- Process:         Voxel Stream Interface
    -- Input(s):        aclk
    -- Output(s):       Voxel Delay Registers
    -- Description:     Voxel Stream Interface Delay to Align to BRAM
    process (aclk)
    begin
        if (rising_edge(aclk)) then
            REG_VOXELS_tdata            <= M_AXIS_MM2S_VOXELS_tdata;
            REG_VOXELS_tlast            <= M_AXIS_MM2S_VOXELS_tlast;
            REG_VOXELS_tvalid           <= M_AXIS_MM2S_VOXELS_tvalid;
        end if;
    end process;


    ------------------------------------------------------------------------------------------------------------------------
    -- Block Memory State Machine
    ------------------------------------------------------------------------------------------------------------------------
    ------------------------------------------------------------------------------------------------------------------------
    -- Process:         BRAM Data
    -- Input(s):        aclk
    -- Output(s):       BRAM Data In Port
    -- Description:     Save results to BRAM
    process (aclk)
    begin
        if (falling_edge(aclk)) then
            case post_comp_valid(0) is
                when X"02" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(0);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(0);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(0);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(0);
                    BRAMP0S0                                 <= r_value(0);
                when X"04" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(1);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(1);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(1);
```

```
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(1);
                    BRAMP0S0                                 <= r_value(1);
                when X"06" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(2);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(2);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(2);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(2);
                    BRAMP0S0                                 <= r_value(2);
                when X"08" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(3);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(3);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(3);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(3);
                    BRAMP0S0                                 <= r_value(3);
                when X"0A" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(4);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(4);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(4);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(4);
                    BRAMP0S0                                 <= r_value(4);
                when X"0C" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(5);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(5);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(5);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(5);
                    BRAMP0S0                                 <= r_value(5);
                when X"0E" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(6);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(6);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(6);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(6);
                    BRAMP0S0                                 <= r_value(6);
                when X"10" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= r_value(7);
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(7);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(7);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(7);
                    BRAMP0S0                                 <= r_value(7);
                when X"00" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= X"05555555";
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(0);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(0);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(0);
                    BRAMP0S0                                 <= X"05555555";
                when X"12" =>
                    RESULT_BRAM_PORT_0_din(31 downto 0)      <= X"55555555";
                    RESULT_BRAM_PORT_0_din(34 downto 32)     <= r_value_low_index(0);
                    RESULT_BRAM_PORT_0_din(36 downto 35)     <= r_value_grp_index(0);
                    RESULT_BRAM_PORT_0_din(63 downto 37)     <= r_value_index(0);
                    BRAMP0S0                                 <= X"55555555";
                when others =>
            end case;
        end if;
    end process;

    ------------------------------------------------------------------------------------------------------------------------
    -- Process:         BRAM Write Enable
    -- Input(s):        aclk
    -- Output(s):       BRAM Data WE Port
    -- Description:     Save results to BRAM
```

```
    process (aclk)
    begin
        if (falling_edge(aclk)) then
            if (got_data(0) = '1') then
                case post_comp_valid(0) is
                    when X"03" =>
                        if (r_capture(0) = '1') then
                            RESULT_BRAM_PORT_0_we       <= X"FF";
                            WR(0) <= '1';
                        else
                            RESULT_BRAM_PORT_0_we       <= X"00";
                            WR(0) <= '0';
                        end if;
                    when X"05" =>
                        if (r_capture(1) = '1') then
                            RESULT_BRAM_PORT_0_we       <= X"FF";
                            WR(0) <= '1';
                        else
                            RESULT_BRAM_PORT_0_we       <= X"00";
                            WR(0) <= '0';
                        end if;
                    when X"07" =>
                        if (r_capture(2) = '1') then
                            RESULT_BRAM_PORT_0_we       <= X"FF";
                            WR(0) <= '1';
                        else
                            RESULT_BRAM_PORT_0_we       <= X"00";
                            WR(0) <= '0';
                        end if;
                    when X"09" =>
                        if (r_capture(3) = '1') then
                            RESULT_BRAM_PORT_0_we       <= X"FF";
                            WR(0) <= '1';
                        else
                            RESULT_BRAM_PORT_0_we       <= X"00";
                            WR(0) <= '0';
                        end if;
                    when X"0B" =>
                        if (r_capture(4) = '1') then
                            RESULT_BRAM_PORT_0_we       <= X"FF";
                            WR(0) <= '1';
                        else
                            RESULT_BRAM_PORT_0_we       <= X"00";
                            WR(0) <= '0';
                        end if;
                    when X"0D" =>
                        if (r_capture(5) = '1') then
                            RESULT_BRAM_PORT_0_we       <= X"FF";
                            WR(0) <= '1';
                        else
                            RESULT_BRAM_PORT_0_we       <= X"00";
                            WR(0) <= '0';
                        end if;
                    when X"0F" =>
                        if (r_capture(6) = '1') then
                            RESULT_BRAM_PORT_0_we       <= X"FF";
                            WR(0) <= '1';
                        else
                            RESULT_BRAM_PORT_0_we       <= X"00";
                            WR(0) <= '0';
```

```
                    end if;
                when X"11" =>
                    if (r_capture(7) = '1') then
                        RESULT_BRAM_PORT_0_we       <= X"FF";
                        WR(0) <= '1';
                    else
                        RESULT_BRAM_PORT_0_we       <= X"00";
                        WR(0) <= '0';
                    end if;
                when others =>
                    RESULT_BRAM_PORT_0_we       <= X"00";
                    WR(0) <= '0';
            end case;
        else
            RESULT_BRAM_PORT_0_we       <= X"00";
            WR(0) <= '0';
        end if;
    end if;
end process;

-------------------------------------------------------------------------------------------------------------------------
-- Process:         BRAM Address Generator
-- Input(s):        aclk
-- Output(s):       BRAM Data WE Port
-- Description:     Save results to BRAM
process (aclk)
begin
    if (falling_edge(aclk)) then
        if axi_aresetn = '0' or system_reset = '1' or got_data(0) = '0' then
            result_address                  <= "0000000000000000";
        else
            case post_comp_valid(0) is
                when X"04" =>
                    if (r_capture(0) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"06" =>
                    if (r_capture(1) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"08" =>
                    if (r_capture(2) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"0A" =>
                    if (r_capture(3) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"0C" =>
                    if (r_capture(4) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"0E" =>
                    if (r_capture(5) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when X"10" =>
                    if (r_capture(6) = '1') then
                        result_address              <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
```

```
                when X"12" =>
                    if (r_capture(7) = '1') then
                        result_address             <= std_logic_vector(unsigned(result_address) + 1);
                    end if;
                when others =>
            end case;
        end if;
    end if;

    RESULT_BRAM_PORT_0_addr(2 downto 0)   <= "000";
    RESULT_BRAM_PORT_0_addr(19 downto 3)  <= result_address;
    RESULT_BRAM_PORT_0_addr(31 downto 20) <= X"000";

    SEEDS_BRAM_PORT_0_addr(5 downto 0)  <= "000000";
    SEEDS_BRAM_PORT_0_addr(17 downto 6) <= seed_count;
    SEEDS_BRAM_PORT_0_addr(31 downto 18) <= "00000000000000";
end process;


-------------------------------------------------------------------------------------------------------------------------
-- Process:         Internal Voxel Data Counter and Accumulator Last Signal - Multiplier Valid
-- Input(s):        aclk
-- Output(s):       Voxel Counter
-- Description:     Counts voxels until reset
process (aclk, voxel_count)
begin
    if (rising_edge(aclk)) then
        if  (axi_aresetn = '0' or system_reset = '1') then
            seed_count          <= X"000";
        elsif ((seed_count = N_SAMPLES) or (seed_count = sample_count_1(0)) or (seed_count = sample_count_2(0))) and (M_AXIS_MM2S_VOXELS_tvalid = '1') then
            seed_count          <= std_logic_vector(unsigned(seed_count) + 1);
        elsif (seed_count > N_SAMPLES) and (M_AXIS_MM2S_VOXELS_tvalid = '1') then
            seed_count          <= X"000";
        elsif M_AXIS_MM2S_VOXELS_tvalid = '1' then
            seed_count          <= std_logic_vector(unsigned(seed_count) + 1);
        end if;
    end if;
end process;

-------------------------------------------------------------------------------------------------------------------------
-- Process:         R-Value Index (Voxel Index) Storage
-- Input(s):        aclk
-- Output(s):       R-Value Index
-- Description:     Set voxel index based upon even/odd (even is voxel #1, odd is voxel #2)
process (aclk, voxel_count)
begin
    if (rising_edge(aclk)) then
        r_value_index(0)    <= VOXELINDEX(0);
        r_value_index(1)    <= VOXELINDEX(1);
        r_value_index(2)    <= VOXELINDEX(0);
        r_value_index(3)    <= VOXELINDEX(1);
        r_value_index(4)    <= VOXELINDEX(0);
        r_value_index(5)    <= VOXELINDEX(1);
        r_value_index(6)    <= VOXELINDEX(0);
        r_value_index(7)    <= VOXELINDEX(1);
    end if;
end process;


-------------------------------------------------------------------------------------------------------------------------
```

```
    -- Loop through all "R" cores
    ------------------------------------------------------------------------------------------------------------------------
    RGEN : for r_index in 0 to (NUM_R_CORES - 1) generate
        --------------------------------------------------------------------------------------------------------------------
        -- Process:         "r" Divider State Machine
        -- Input(s):        aclk, Voxel Count
        -- Output(s):       BRAM Address
        -- Description:     Updates BRAM Address, Reset on start or increments on specific voxel count
        process (aclk, voxel_count)
        begin
            if (falling_edge(aclk)) then
                case voxel_count_acc(r_index) is
                    when X"01E" =>
                        a_dividend_data(r_index)    <=  RESULTS_COV(r_index);
                        b_divisor_data(r_index)     <=  RESULTS_SD(r_index);
                        a_dividend_valid(r_index)   <= '1';
                        b_divisor_valid(r_index)    <= '1';
                    when X"032" =>
                        a_dividend_data(r_index)    <=  RESULTS_COV(r_index + 8);
                        b_divisor_data(r_index)     <=  RESULTS_SD(r_index + 8);
                        a_dividend_valid(r_index)   <= '1';
                        b_divisor_valid(r_index)    <= '1';
                    when X"046" =>
                        a_dividend_data(r_index)    <=  RESULTS_COV(r_index + 16);
                        b_divisor_data(r_index)     <=  RESULTS_SD(r_index + 16);
                        a_dividend_valid(r_index)   <= '1';
                        b_divisor_valid(r_index)    <= '1';
                    when X"05A" =>
                        a_dividend_data(r_index)    <=  RESULTS_COV(r_index + 24);
                        b_divisor_data(r_index)     <=  RESULTS_SD(r_index + 24);
                        a_dividend_valid(r_index)   <= '1';
                        b_divisor_valid(r_index)    <= '1';
                    when others =>
                        a_dividend_valid(r_index)   <= '0';
                        b_divisor_valid(r_index)    <= '0';
                end case;
            end if;
        end process;

        ----------------------------------------------------------------------------------------------------------------
        --! U_DIV: floating_point_div
        --! Clock on positive edge of ACLK
        --! Perform division of Covar and SD Term
        ----------------------------------------------------------------------------------------------------------------
        U_DIV : floating_point_div
        Port Map
        (
            aclk                        => aclk,
            s_axis_a_tvalid             => a_dividend_valid(r_index),
            s_axis_a_tready             => open,
            s_axis_a_tdata              => a_dividend_data(r_index),
            s_axis_b_tvalid             => b_divisor_valid(r_index),
            s_axis_b_tready             => open,
            s_axis_b_tdata              => b_divisor_data(r_index),
            m_axis_result_tvalid        => r_quotient_valid(r_index),
            m_axis_result_tready        => '1',
            m_axis_result_tdata         => r_quotient_data(r_index)
        );

        ----------------------------------------------------------------------------------------------------------------
```

```
        --! U_COMP: floating_point_comp
        --! Clock on positive edge of ACLK
        --! Perform comparison of "r" value ("A") to r' ("B") set to application
        ------------------------------------------------------------------------------------------------------------------
        U_COMP : floating_point_comp
        Port Map
        (
            aclk                        => aclk,
            s_axis_a_tvalid             => r_quotient_valid(r_index),
            s_axis_a_tready             => open,
            s_axis_a_tdata              => r_quotient_data(r_index),
            s_axis_b_tvalid             => b_comp_valid,
            s_axis_b_tready             => open,
            s_axis_b_tdata              => b_comp_data,
            m_axis_result_tvalid        => r_comp_valid(r_index),
            m_axis_result_tready        => '1',
            m_axis_result_tdata         => r_comp_data(r_index)
        );

        ------------------------------------------------------------------------------------------------------------------
        --! U_COMPMAX: floating_point_comp
        --! Clock on positive edge of ACLK
        --! Perform comparison of "r" value ("A") to r' ("B") set to application (i.e. "r" <= 1.0)
        ------------------------------------------------------------------------------------------------------------------
        U_COMPLTE : floating_point_complte
        Port Map
        (
            aclk                        => aclk,
            s_axis_a_tvalid             => r_quotient_valid(r_index),
            s_axis_a_tready             => open,
            s_axis_a_tdata              => r_quotient_data(r_index),
            s_axis_b_tvalid             => '1',
            s_axis_b_tready             => open,
            s_axis_b_tdata              => X"3F800000",
            m_axis_result_tvalid        => r_complte_valid(r_index),
            m_axis_result_tready        => '1',
            m_axis_result_tdata         => r_complte_data(r_index)
        );



        ------------------------------------------------------------------------------------------------------------------
        -- Process:         Capture "r" value
        -- Input(s):        aclk
        -- Output(s):       Captured "r" data
        -- Description:     Save results to register
        process (aclk)
        begin
            if (rising_edge(aclk)) then
                if (r_quotient_valid(r_index) = '1') then
                    r_value(r_index)            <= r_quotient_data(r_index);
                end if;
            end if;
        end process;

        ------------------------------------------------------------------------------------------------------------------
        -- Process:         Capture "r" value index - R-Index (group)
        -- Input(s):        aclk
        -- Output(s):       Captured "r" data
        -- Description:     Save results to register
```

```
        process (aclk)
        begin
            if (rising_edge(aclk)) then
                if axi_aresetn = '0' or system_reset = '1' or accumulator_last_out(r_index) = '1' then
                    r_value_grp_index(r_index)                  <= "11";
                elsif (r_quotient_valid(r_index) = '1') then
                    r_value_grp_index(r_index)                  <= std_logic_vector(unsigned(r_value_grp_index(r_index)) + 1);
                end if;
            end if;
        end process;

        ------------------------------------------------------------------------------------------------------------
        -- Process:         Capture "r" value
        -- Input(s):        aclk
        -- Output(s):       Captured "r" data
        -- Description:     Save results to register
        process (aclk)
        begin
            if (rising_edge(aclk)) then
                if axi_aresetn = '0' or system_reset = '1' or got_data(0) = '0' then
                    r_capture(r_index)  <= '0';
                elsif (r_comp_valid(r_index) = '1') and (r_complte_valid(r_index) = '1') then
                    if (r_comp_data(r_index)(0) = '1') and (r_complte_data(r_index)(0) = '1') then
                        r_capture(r_index)  <= '1';
                    else
                        r_capture(r_index)  <= '0';
                    end if;
                end if;
            end if;
        end process;

        ------------------------------------------------------------------------------------------------------------
        -- Process:         Post valid clock counter
        -- Input(s):        aclk
        -- Output(s):       Captured "r" data
        -- Description:     Save results to register
        process (aclk)
        begin
            if (rising_edge(aclk)) then
                if (r_comp_valid(r_index) = '1') then
                    post_comp_valid(r_index)    <= X"00";
                elsif (post_comp_valid(r_index) < X"80") then
                    post_comp_valid(r_index)    <= std_logic_vector(unsigned(post_comp_valid(r_index)) + 1);
                end if;
            end if;
        end process;

    end generate RGEN;


    ------------------------------------------------------------------------------------------------------------------
    -- Loop through all voxels
    ------------------------------------------------------------------------------------------------------------------
    VOXELIDXGEN : for voxel_index in 0 to (NUM_VOXELS - 1) generate
        -- Capture voxel index
        process (aclk, sample_count_3, voxel_count)
        begin
            if (rising_edge(aclk)) then
                if voxel_count(voxel_index) = sample_count_3(voxel_index) and REG_VOXELS_tvalid = '1' then
                    VOXELINDEX(voxel_index)       <= REG_VOXELS_tdata(((voxel_index * DATA_WIDTH_BITS) + 31)  downto (voxel_index * DATA_WIDTH_BITS));
```

```
            end if;
        end if;
    end process;
end generate VOXELIDXGEN;

-------------------------------------------------------------------------------------------------------------------
-- Loop through all seeds and voxels
-------------------------------------------------------------------------------------------------------------------
SEEDGEN : for seed_index in 0 to (NUM_SEEDS - 1) generate
    VOXELGEN : for voxel_index in 0 to (NUM_VOXELS - 1) generate
        -------------------------------------------------------------------------------------------------------------------
        --! U_MUL: floating_point_mult
        --! Clock on positive edge of ACLK
        --! Perform multiplication of Seed and Voxel
        -------------------------------------------------------------------------------------------------------------------
        U_MUL : floating_point_mul
        Port Map
        (
            aclk                        => aclk,
            s_axis_a_tvalid             => REG_VOXELS_tvalid,
            s_axis_a_tready             => M_AXIS_MM2S_VOXELS_ready((seed_index * NUM_VOXELS) + voxel_index),
            s_axis_a_tdata              => REG_VOXELS_tdata(((voxel_index * DATA_WIDTH_BITS) + 31)  downto (voxel_index * DATA_WIDTH_BITS)),
            s_axis_a_tlast              => REG_VOXELS_tlast,
            s_axis_b_tvalid             => '1',
            s_axis_b_tready             => open,
            s_axis_b_tdata              => SEEDS_BRAM_PORT_0_dout(((seed_index * DATA_WIDTH_BITS) + 31)  downto (seed_index * DATA_WIDTH_BITS)),
            s_axis_b_tlast              => '0',
            m_axis_result_tvalid        => multiplier_valid((seed_index * NUM_VOXELS) + voxel_index),
            m_axis_result_tready        => accumulator_ready((seed_index * NUM_VOXELS) + voxel_index),
            m_axis_result_tdata         => multiplier_result((seed_index * NUM_VOXELS) + voxel_index),
            m_axis_result_tlast         => multiplier_last((seed_index * NUM_VOXELS) + voxel_index)
        );

        -------------------------------------------------------------------------------------------------------------------
        --! U_ACC: floating_point_acc
        --! Clock on positive edge of ACLK
        --! Perform accumulation of Seed and Voxel
        -------------------------------------------------------------------------------------------------------------------
        U_ACC : floating_point_acc
            Port Map
            (
                aclk                        => aclk,
                aresetn                     => accumulator_reset((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tvalid             => multiplier_valid((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tready             => accumulator_ready((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tdata              => multiplier_result((seed_index * NUM_VOXELS) + voxel_index),
                s_axis_a_tlast              => accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index),
                m_axis_result_tvalid        => accumulator_valid((seed_index * NUM_VOXELS) + voxel_index),
                m_axis_result_tready        => '1',
                m_axis_result_tdata         => RESULTS_0((seed_index * NUM_VOXELS) + voxel_index),
                m_axis_result_tlast         => accumulator_last_out((seed_index * NUM_VOXELS) + voxel_index)
            );

        -- Capture results - Result Bank Voxel #0 or Voxel #1
        process (aclk, accumulator_last_out)
        begin
            if (rising_edge(aclk)) then
                if (accumulator_last_out_1((seed_index * NUM_VOXELS) + voxel_index) = '1' and
                    accumulator_last_out_2((seed_index * NUM_VOXELS) + voxel_index) = '0' and
                    accumulator_last_out_3((seed_index * NUM_VOXELS) + voxel_index) = '0') then
```

```
                RESULTS_COV((seed_index * NUM_VOXELS) + voxel_index)    <= RESULTS_0((seed_index * NUM_VOXELS) + voxel_index);
            end if;
        end if;
    end process;


    process (aclk)
    begin
        if (rising_edge(aclk)) then
            sample_count_1((seed_index * NUM_VOXELS) + voxel_index)          <=  std_logic_vector(unsigned(N_SAMPLES) + 1);
            sample_count_2((seed_index * NUM_VOXELS) + voxel_index)          <=  std_logic_vector(unsigned(N_SAMPLES) + 2);
            sample_count_3((seed_index * NUM_VOXELS) + voxel_index)          <=  std_logic_vector(unsigned(N_SAMPLES) - 4);
        end if;
    end process;

    ----------------------------------------------------------------------------------------------------------
    -- Process:         Last Out Delay Registers
    -- Input(s):        aclk
    -- Output(s):       Last Outs
    -- Description:     4-stage last out register
    process (aclk, accumulator_last_out, accumulator_last_out_1, accumulator_last_out_2, accumulator_last_out_3)
    begin
        if (falling_edge(aclk)) then
            accumulator_last_out_1((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out((seed_index * NUM_VOXELS) + voxel_index);
            accumulator_last_out_2((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out_1((seed_index * NUM_VOXELS) + voxel_index);
            accumulator_last_out_3((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out_2((seed_index * NUM_VOXELS) + voxel_index);
            accumulator_last_out_4((seed_index * NUM_VOXELS) + voxel_index)  <= accumulator_last_out_3((seed_index * NUM_VOXELS) + voxel_index);
        end if;
    end process;

    ----------------------------------------------------------------------------------------------------------
    -- Process:         Internal Voxel Data Counter and Accumulator Last Signal - Multiplier Valid
    -- Input(s):        aclk
    -- Output(s):       Voxel Counter
    -- Description:     Counts voxels until reset
    process (aclk, voxel_count)
    begin
        if (rising_edge(aclk)) then
            -- If voxel count greater than spec. value then reset
            if  (axi_aresetn = '0' or system_reset = '1') then
                accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '0';-- CHANGE from zero
                got_data((seed_index * NUM_VOXELS) + voxel_index) <= '0';
                voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= X"000";
            elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) = N_SAMPLES) and
                  (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
                accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '1';
                got_data((seed_index * NUM_VOXELS) + voxel_index) <= '1';
                voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
            elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) = sample_count_1((seed_index * NUM_VOXELS) + voxel_index)) and
                  (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
                accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '1';
                voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
            elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) = sample_count_2((seed_index * NUM_VOXELS) + voxel_index)) and
                  (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
                accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '1';
                got_data((seed_index * NUM_VOXELS) + voxel_index) <= '1';
                voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
                RESULTS_SD((seed_index * NUM_VOXELS) + voxel_index)          <= multiplier_result((seed_index * NUM_VOXELS) + voxel_index);
            elsif (voxel_count((seed_index * NUM_VOXELS) + voxel_index) > N_SAMPLES) and (multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
                accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '0';
```

```
                voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= X"000";
            elsif multiplier_valid((seed_index * NUM_VOXELS) + voxel_index) = '1' then
                accumulator_last_in((seed_index * NUM_VOXELS) + voxel_index) <= '0';
                voxel_count((seed_index * NUM_VOXELS) + voxel_index)         <= std_logic_vector(unsigned(voxel_count((seed_index * NUM_VOXELS) + voxel_index)) + 1);
            end if;
        end if;
    end process;


    ---------------------------------------------------------------------------------------------------------------
    -- Process:         Accumulator Reset
    -- Input(s):        aclk
    -- Output(s):       Reset
    -- Description:      Accumulator Reset
    process (aclk, voxel_count)
    begin
        if (rising_edge(aclk)) then
            if  (axi_aresetn = '0' or system_reset = '1') then
                accumulator_reset((seed_index * NUM_VOXELS) + voxel_index) <= '0';
            else
                accumulator_reset((seed_index * NUM_VOXELS) + voxel_index) <= '1';
            end if;
        end if;
    end process;

    ---------------------------------------------------------------------------------------------------------------
    -- Process:         Internal Voxel Data Counter and Accumulator Last Signal - Accumulator Valid
    -- Input(s):        aclk
    -- Output(s):       Voxel Counter
    -- Description:     Counts voxels until reset
    process (aclk, voxel_count_acc)
    begin
        if (rising_edge(aclk)) then
            -- If voxel count greater than spec. value then reset
            if  (axi_aresetn = '0' or system_reset = '1') then
                voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= X"000";
            elsif ((voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) = N_SAMPLES) or
                   (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) = sample_count_1((seed_index * NUM_VOXELS) + voxel_index)) or
                   (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) = sample_count_2((seed_index * NUM_VOXELS) + voxel_index))) and
                  (accumulator_valid((seed_index * NUM_VOXELS) + voxel_index) = '1') then
                voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= std_logic_vector(unsigned(voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index))
                + 1);
            elsif (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) > N_SAMPLES) and (accumulator_last_out_4((seed_index * NUM_VOXELS) + voxel_index) =
            '1') then
                voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= X"000";
            elsif (voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index) < X"FFF") and (got_data((seed_index * NUM_VOXELS) + voxel_index) = '1') then
                voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index)     <= std_logic_vector(unsigned(voxel_count_acc((seed_index * NUM_VOXELS) + voxel_index))
                + 1);
            end if;
        end if;
    end process;

    -- For the first index set all bits to initial index value
    I0: if ((seed_index * NUM_VOXELS) + voxel_index) = 0 generate
        M_AXIS_MM2S_VOXELS_tready_T(0)  <= M_AXIS_MM2S_VOXELS_ready(0);
    end generate I0;

    -- Any other index and/or with the previous value
    IX: if ((seed_index * NUM_VOXELS) + voxel_index) > 0 generate
        M_AXIS_MM2S_VOXELS_tready_T((seed_index * NUM_VOXELS) + voxel_index) <= M_AXIS_MM2S_VOXELS_tready_T(((seed_index * NUM_VOXELS) + voxel_index) - 1) and
```

```
                M_AXIS_MM2S_VOXELS_ready((seed_index * NUM_VOXELS) + voxel_index);
            end generate IX;
        end generate VOXELGEN;
    end generate SEEDGEN;

    ------------------------------------------------------------------------------------------------------------------
    -- And results for output (i.e. get last index from gen)
    ------------------------------------------------------------------------------------------------------------------
    M_AXIS_MM2S_VOXELS_tready           <=  M_AXIS_MM2S_VOXELS_tready_T(((NUM_VOXELS * NUM_SEEDS)-1));
end RTL;
```

APPENDIX D

SCRIPTS, SOFTWARE, AND FIRMWARE

```
//---------------------------------------------------------------------------
// High-Performance Correlation and Mapping Engine for
// Rapid Generating Brain Connectivity Networks from Big fMRI Data
//
//    File Name:    cca_main.c
//      Version:    1.0.0
//       Author:    John Lusher II, P.E.
//                  Texas A&M University
//         Date:    July 15, 2017
//  Description:    fMRI Brain Connectivity Maps Processing Engine - main function
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Revision(s):     Date:                       Description:
// v1.0.0           July 20, 2017               Initial Release
// v1.1.0           Sep 20, 2017                PoC System Test Release
// v1.2.0           Nov 25, 2017                HPCME System  Test Release
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
//  Header Files
//---------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#include "dma.h"
#include "bram.h"
#include "platform.h"
#include "xaxidma.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "xgpio.h"

#include "xparameters.h"
#include "xscugic.h"
#include "xaxidma.h"
#include "xil_printf.h"
#include "xtime_l.h"

#include "sleep.h"

#include "cca_main.h"
#include "cca_processing.h"
#include "console_communications.h"


//---------------------------------------------------------------------------
// Definitions
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Global variables
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// Device Instances
//---------------------------------------------------------------------------
```

```
                                                    //------------------------
XAxiDma axi_dma_seed;                               // DMA Instance for Seeds
XAxiDma axi_dma_voxel;                              // DMA Instance for Voxels
XAxiDma axi_dma_results;                            // DMA Instance for CoVar Results
XAxiDma axi_dma_rval;                               // DMA Instance for R-Value / SD Value
XGpio InterfaceGPIO;                                // GPIO Instance
XGpio ControlGPIO;                                  // GPIO Instance #2 (In / Out)
                                                    //------------------------


                                                    //------------------------
XDmaPs  dma_results;                                // DMA Results
volatile int Checked[XDMAPS_CHANNELS_PER_DEV];      // DMA check for results
XDmaPs_Cmd DmaCmd;                                  // DMA Command
                                                    //------------------------


                                                    //------------------------
XBram axi_bram_seed;                                // BRAM Instance for Seeds
XBram axi_bram_result_voxel_0;                      // BRAM Instance for Voxel #0 Results
XBram axi_bram_result_voxel_1;                      // BRAM Instance for Voxel #1 Results
XScuGic axi_intc;                                   // Interrupt Instance
                                                    //------------------------


                                                    //------------------------
int mm2s_done_0;                                    // Flags which get set by DMA ISR
int mm2s_done_1;                                    //
int dma_err_0;                                      // Error in DMA flag 0
int dma_err_1;                                      // Error in DMA flag 1


int r_dma_out_done;                                 // DMA "R" Stream Out (Read)
int r_dma_in_done;                                  // DMA "R" Stream Out (Write)
int result_dma_done;                                // DMA Result
int dma_r_err_0;                                    // Error in R DMA flag 0
int dma_r_err_1;                                    // Error in R DMA flag 1
int dma_result_err;                                 // Error in Result DMA flag

float *inp_stream_voxels;
float *inp_seed_values;
float *inp_r_values_0;
char buffer_array[256];

int process_data = FALSE;
int process_task = FALSE;
int current_task = -1;

u32 GPIO_mask = 0x00000000;                         // GPIO Mask
u32 r_prime_mask = 0x00000000;                      // R' GPIO

int assigned_system = 0;
int number_of_samples;
int voxel_offset = 0;
int voxels_per_group;
int voxels_group = 0;
int voxels_per_set = 0;
int num_voxels = 0;
int num_seeds = 16;
int num_voxel_groups = 0;
int seeds_per_group = 0;
float r_prime = -1.0;
```

```
int voxel_group = -1;
int seed_group = -1;

int start_voxel = 0;
int num_process_groups = 0;

struct tasks tasks[256000];                                     // Max tasks are 256K
tasks_t tasklist;
int number_tasks = 0;


int prev_task = -1;


//-----------------------------------------------------------------------------
//  Routine:    print_header
//   Inputs:    none
//  Outputs:    none
//  Purpose:    Prints the header for the application
//              print information such as university, project, version, build #,
//              and any other relevant information
//-----------------------------------------------------------------------------
void print_header()
{
    xil_printf("\n\r-------------------------------------------------------------------------------------------\n\r");
    xil_printf("High-Performance Correlation and Mapping Engine for Rapid Generating\n\r");
    xil_printf("Brain Connectivity Networks from Big fMRI Data\n\n\r");
    xil_printf("Version: v%d.%d.%d.%d\n\r", VERSION_MAJOR, VERSION_MINOR, VERSION_REV, VERSION_BUILD);
    xil_printf("-------------------------------------------------------------------------------------------\n\r");
    xil_printf("Texas A&M University\n\r");
    xil_printf("Department of Electrical and Computer Engineering\n\r");
    xil_printf("3128 TAMU\n\r");
    xil_printf("College Station, TX 77843-3128\n\r");
    xil_printf("-------------------------------------------------------------------------------------------\n\r");
    xil_printf("Console Interface\n\r");
    xil_printf("-------------------------------------------------------------------------------------------\n\n\r");
}

//-----------------------------------------------------------------------------
//  Routine:    main
//   Inputs:    none
//  Outputs:    result (zero = success, non-zero = error)
//  Purpose:    Test the HLS component
//-----------------------------------------------------------------------------
int main()
{
    //-------------------------------------------------------------------------
    // Variables for status, indices, and other housekeeping
    //-------------------------------------------------------------------------
    int status = 0;
    float setup_time = 0.0;


    //-------------------------------------------------------------------------
    // Union to convert integer to/from floating point (single)
    //-------------------------------------------------------------------------
    union
    {
        unsigned int ival;
```

```
        float fval;
    } converter;


    //----------------------------------------------------------------------
    // INITALIZE SYSTEM
    //----------------------------------------------------------------------
    //----------------------------------------------------------------------
    // Setup UART and enable caching
    //----------------------------------------------------------------------
    init_platform();

    //----------------------------------------------------------------------
    // Print header
    //----------------------------------------------------------------------
    print_header();

    //----------------------------------------------------------------------
    // Allocate memory
    //----------------------------------------------------------------------
    unsigned int dma_size_in_voxels = (140*1024*1024) * sizeof(int);
    inp_stream_voxels = malloc(dma_size_in_voxels);
    if (inp_stream_voxels == NULL)
    {
        xil_printf("\rMEMORY ALLOCATION ERROR INPUT VOXEL DMA\n\r");
        return XST_FAILURE;
    }

    // Allocate a max of 128 MB for seeds
    inp_seed_values = malloc(128*1024*1024);
    if (inp_seed_values == NULL)
    {
        xil_printf("\rMEMORY ALLOCATION ERROR SEED\n\r");
        return XST_FAILURE;
    }

    // Allocate a max of 64 MB for results
    inp_r_values_0 = malloc(64*1024*1024);
    if (inp_r_values_0 == NULL)
    {
        xil_printf("\rMEMORY ALLOCATION ERROR RESULTS DMA\n\r");
        return XST_FAILURE;
    }
    xil_printf("DONE\r\n");


    //----------------------------------------------------------------------
    // Initialize drivers, return error if not successful
    //----------------------------------------------------------------------
    status = init_drivers();
    if (status != XST_SUCCESS)
    {
        xil_printf("Driver initialization failed!\n\r");
        return XST_FAILURE;
    }

    //----------------------------------------------------------------------
    // BRAM Initialization
    //----------------------------------------------------------------------
    // Seeds
```

```
    status = bram_init_seed();
    if (status != XST_SUCCESS)
    {
        xil_printf("Seed BRAM driver initialization failed!\n\r");
        return XST_FAILURE;
    }

    //---------------------------------------------------------------------
    // GPIO Initialization
    //---------------------------------------------------------------------
    status = XGpio_Initialize(&InterfaceGPIO, XPAR_AXI_GPIO_0_DEVICE_ID);
    if (status != XST_SUCCESS)                                      // If not configured then exit
    {                                                               //
          xil_printf("GPIO initialization failed, status %d\r\n", status);
          return XST_FAILURE;                                       //
    }                                                               //


    //---------------------------------------------------------------------
    // GPIO Initialization
    //---------------------------------------------------------------------
    status = XGpio_Initialize(&ControlGPIO, XPAR_AXI_GPIO_1_DEVICE_ID);
    if (status != XST_SUCCESS)                                      // If not configured then exit
    {                                                               //
          xil_printf("GPIO #2 initialization failed, status %d\r\n", status);
          return XST_FAILURE;                                       //
    }                                                               //


    //---------------------------------------------------------------------
    // Blink LEDs on startup, simple hardware verification
    //---------------------------------------------------------------------
    for (GPIO_mask = 0; GPIO_mask < 0x10; GPIO_mask++)              //
    {                                                               //
        XGpio_DiscreteClear(&InterfaceGPIO, 1, ~GPIO_mask);         // Clear bits
        XGpio_DiscreteSet(&InterfaceGPIO, 1, GPIO_mask);            // Set data to GPIO
        usleep(250000);                                             // Sleep 1/4 sec
    }                                                               //

    xil_printf("Setup communications - standby....\r\n");
    sleep(10);

    //---------------------------------------------------------------------
    // Set "N" Samples for core (default valuse, setup by task controller)
    //---------------------------------------------------------------------
    GPIO_mask = 0x3FD;                                              // Test "N" (1023)
    XGpio_DiscreteClear(&InterfaceGPIO, 1, ~GPIO_mask);             // Clear bits
    XGpio_DiscreteSet(&InterfaceGPIO, 1, GPIO_mask);                // Set data to GPIO

    GPIO_mask |= 0x80000000;                                        // Set Reset
    XGpio_DiscreteClear(&InterfaceGPIO, 1, ~GPIO_mask);             // Clear bits
    XGpio_DiscreteSet(&InterfaceGPIO, 1, GPIO_mask);                // Set data to GPIO
    usleep(500000);                                                 // Sleep 1/2 sec

    GPIO_mask &= 0x00000FFF;                                        // Clear Reset
    XGpio_DiscreteClear(&InterfaceGPIO, 1, ~GPIO_mask);             // Clear bits
    XGpio_DiscreteSet(&InterfaceGPIO, 1, GPIO_mask);                // Set data to GPIO
    usleep(500000);                                                 // Sleep 1/2 sec

    //---------------------------------------------------------------------
```

```
    // Set r' for core
    //------------------------------------------------------------------------
    converter.fval = r_prime;
    r_prime_mask = converter.ival;
    XGpio_DiscreteClear(&ControlGPIO, 2, ~r_prime_mask);            // Clear bits
    XGpio_DiscreteSet(&ControlGPIO, 2, r_prime_mask);               // Set data to GPIO


    //------------------------------------------------------------------------
    // Console Communications (UART 1 - 115200 bps, 8N1)
    //------------------------------------------------------------------------
    status = console_comm_init(&axi_intc);                          // Console Communications initialization
    if (status != XST_SUCCESS)                                      // If initialization fails then exit
    {                                                               //
          xil_printf("Console Communications initialization failed\r\n");
          return XST_FAILURE;                                       //
    }                                                               //

    //------------------------------------------------------------------------
    // System ready
    xil_printf("System ready\r\n\n");


    // Set ch 1 to all inputs
    XGpio_SetDataDirection(&ControlGPIO, 1, 0xFFFFFFFF);


    //------------------------------------------------------------------------
    // Infinite loop
    //------------------------------------------------------------------------
    while (TRUE)
    {

        // Parse communications and process from console & TCP_IP if needed
        process_console_communications();


        //------------------------------------------------------------------------
        // Process data (DMA Process)
        if (process_data)
        {
            XTime start_time_set, end_time_set;
            int status_1;

            int group_size, group_index, num_groups;
            int result_addr;
            int offset_start;
            int end_addr, end_addr_prev;
            float process_time = 0.0;

            // Start results at zero
            int current_index = 0;
            int start_voxel = 0;

            // Start task with header data
            printf("%3d\t%5d\t%8d\t%4d\t", voxel_group, seed_group, voxel_offset, number_of_samples);

            // Start Time
            XTime_GetTime(&start_time_set);
```

```
// Set N-2 Samples
GPIO_mask = number_of_samples - 2;                          // Set N-2
XGpio_DiscreteClear(&InterfaceGPIO, 1, ~GPIO_mask);         // Clear bits
XGpio_DiscreteSet(&InterfaceGPIO, 1, GPIO_mask);            // Set data to GPIO

// Determine group size, number of groups from sample size
// Goal is to run as much at a time and collect result,
// 32 allows for 2 voxels to be processed by 16 seeds (32 results per pair)
// leaving a data results of 1K per group of 32 (i.e. 8KB / group)
num_groups = 64;
group_size = (number_of_samples + 2) * num_groups;
offset_start = voxel_offset * (number_of_samples + 2);
end_addr = voxels_per_group * (number_of_samples + 2);
end_addr_prev = end_addr - group_size;

// Start voxel begins at the voxel offset
start_voxel = voxel_offset;

// Loop through all data in groups
for (group_index = offset_start; group_index < end_addr; group_index += group_size)
{

    // Load the voxels and seeds
    #ifdef PRINT_DEBUG
        int end_voxel = (group_index + group_size - 1)/(number_of_samples + 2);
        printf("Start Voxel %d to %d, max: %d  \t-> Start: %d \tEnd: %d\n\r",group_index, group_index + group_size - 1, end_addr, start_voxel, end_voxel);
    #endif
    mm2s_done_1 = 0;

    // Start DMA for group of data
    status_1 = XAxiDma_SimpleTransfer(&axi_dma_voxel, (unsigned int) &inp_stream_voxels[group_index], group_size * sizeof(float), XAXIDMA_DMA_TO_DEVICE);
    if (status_1 != XST_SUCCESS)
    {
        print("Error: Voxel DMA transfer to CCA block failed\n");
        return XST_FAILURE;
    }

    // Wait until done
    while (!mm2s_done_1)
    {
    }

    // Read result address
    result_addr = XGpio_DiscreteRead(&ControlGPIO,1);
    if (((result_addr > 3000) || (group_index >= end_addr_prev)) & (result_addr > 0))
    {
        // Reset flags
        r_dma_out_done = 0;
        r_dma_in_done = 0;
        result_dma_done = 0;
        dma_r_err_0 = 0;
        dma_r_err_1 = 0;
        dma_result_err = 0;

        // Build header (results are 8 bytes / vector)
        current_index = save_results_header(current_index, voxel_group, seed_group, start_voxel, result_addr * 8);

        //---------------------------------------------------------------------
        // Transfer to get "R" value into SDRAM - at location specified
        //---------------------------------------------------------------------
```

```
        // Begin streaming data to the Results DMA, return error if not successful
        //-----------------------------------------------------------------------
        status_1 = XAxiDma_SimpleTransfer(&axi_dma_rval, (unsigned int) &inp_r_values_0[current_index], result_addr * 2 * sizeof(float), XAXIDMA_DEVICE_TO_DMA);
        if (status_1 != XST_SUCCESS)
        {
            print("Error:Result DMA transfer get 'r' #1 failed\n");
            return XST_FAILURE;
        }

        //-----------------------------------------------------------------------
        // Begin streaming data to the Results DMA, return error if not successful
        //-----------------------------------------------------------------------
        status_1 = XAxiDma_SimpleTransfer(&axi_dma_results, 0xC0000000, result_addr * 2 * sizeof(float), XAXIDMA_DMA_TO_DEVICE);
        if (status_1 != XST_SUCCESS)
        {
            print("Error: Result DMA transfer from results #1 failed\n");
            return XST_FAILURE;
        }

        // Wait until done
        while (!r_dma_in_done | !result_dma_done)
        {
        }

        // Increment the result address, there are two entries per vector result
        current_index = current_index + (result_addr * 2);

        // Start voxel is now updated with current position, as the offset counter index resets
        start_voxel = offset_start / (number_of_samples + 2);
    }
    else if (group_index >= end_addr_prev)              // Else, we we at the end BUT, no data, we still need to read the block,
    {                                                   // save the results but NOT add it to the mem
        // Reset flags
        r_dma_out_done = 0;
        r_dma_in_done = 0;
        result_dma_done = 0;
        dma_r_err_0 = 0;
        dma_r_err_1 = 0;
        dma_result_err = 0;

        //-----------------------------------------------------------------------
        // Transfer to get "R" value into SDRAM - at location specified - just a small block of data (32 results)
        //-----------------------------------------------------------------------
        // Begin streaming data to the Results DMA, return error if not successful
        //-----------------------------------------------------------------------
        status_1 = XAxiDma_SimpleTransfer(&axi_dma_rval, (unsigned int) &inp_r_values_0[current_index], 32 * 2 * sizeof(float), XAXIDMA_DEVICE_TO_DMA);
        if (status_1 != XST_SUCCESS)
        {
            print("Error:Result DMA transfer get 'r' #1 failed\n");
            return XST_FAILURE;
        }

        //-----------------------------------------------------------------------
        // Begin streaming data to the Results DMA, return error if not successful
        //-----------------------------------------------------------------------
        status_1 = XAxiDma_SimpleTransfer(&axi_dma_results, 0xC0000000, 32 * 2 * sizeof(float), XAXIDMA_DMA_TO_DEVICE);
        if (status_1 != XST_SUCCESS)
        {
            print("Error: Result DMA transfer from results #1 failed\n");
            return XST_FAILURE;
```

```
                }

                // Wait until done
                while (!r_dma_in_done | !result_dma_done)
                {
                }

                // Start voxel is now updated with current position, as the offset counter index resets
                start_voxel = offset_start / (number_of_samples + 2);
            }
        }

        // Capture end time
        XTime_GetTime(&end_time_set);
        {
            u64 delta_set = (u64)(end_time_set - start_time_set);
            float time_set = (float)delta_set / COUNTS_PER_SECOND;
            process_time = time_set;
        }

        // Capture start time
        XTime_GetTime(&start_time_set);

        // Save results to file, if there are results
        if (current_index > 0)
        {
            write_results((unsigned int)&inp_r_values_0[0], current_index * sizeof(float), assigned_system, voxel_group, seed_group);
        }

        // Capture end time, display result time
        XTime_GetTime(&end_time_set);
        {
            u64 delta_set = (u64)(end_time_set - start_time_set);
            float num_bytes = (float)(current_index * sizeof(float));
            float save_time = (float)delta_set / COUNTS_PER_SECOND;
            float data_rate =  num_bytes / (save_time*1024*1024);
            float total_time = process_time + save_time + setup_time;
            // Footer data, display process time
            printf("%8.5f\t%8.5f\t%8d\t%8.5f\t%8.5f\n\r", process_time, save_time, current_index * sizeof(float), data_rate, total_time);
        }

        process_data = FALSE;
    }

    //---------------------------------------------------------------------
    // Process task list
    if (process_task)
    {
        // Go to next task #
        current_task++;
        if (current_task >= 0)
        {
            // Get next task
            tasklist.voxel_group = tasks[current_task].voxel_group;
            tasklist.seed_group = tasks[current_task].seed_group;
            tasklist.voxel_offset = tasks[current_task].voxel_offset;

            if (current_task >= number_tasks)
            {
                process_task = FALSE;
```

```
            printf("\n\rTask Process Finished - System #%d\n\r", assigned_system);

            // Reset group numbers, force load
            voxel_group = -1;
            seed_group = -1;
        }
        else
        {
            XTime start_time_set, end_time_set;

            // Start time
            XTime_GetTime(&start_time_set);

            // Load voxel data as needed
            if (tasklist.voxel_group != voxel_group)
            {
                voxel_group = tasklist.voxel_group;
                load_voxels(voxel_group);
            }

            // Load seed data as needed
            if (tasklist.seed_group != seed_group)
            {
                int new_major_group = tasklist.seed_group / 1024;
                int current_major_group = seed_group / 1024;
                if ((new_major_group != current_major_group) | (seed_group < 0)) load_seeds(tasklist.seed_group);
                seed_group = tasklist.seed_group;
                txfr_seeds(seed_group);
            }

            // Create main result file (only first task)
            if (current_task == 0)
            {
                // Print Log Header
                printf("System:\tTask:\t# Tasks:\tLoad Time (sec):\tVoxel Group:\tSeed Group:\tOffset:\tSamples:\tProcess Time (sec):\tSave Time (sec):\tResult
                Size (bytes):\tData Rate (MB/sec):\tTotal Time (sec):\n\r");

                // Set R-Prime
                converter.fval = r_prime;
                r_prime_mask = converter.ival;
                XGpio_DiscreteClear(&ControlGPIO, 2, ~r_prime_mask);            // Clear bits
                XGpio_DiscreteSet(&ControlGPIO, 2, r_prime_mask);               // Set data to GPIO


                prev_task = current_task;

                create_result_file(assigned_system,
                                   number_of_samples,
                                   num_voxels,
                                   num_voxel_groups,
                                   voxels_per_group,
                                   voxels_per_set,
                                   num_seeds,
                                   seeds_per_group);
            }

            // Print Task Data
            printf("%d\t%6d\t%6d\t", assigned_system, current_task, number_tasks);

            // Set offset
```

```
                    voxel_offset = tasklist.voxel_offset;

                    XTime_GetTime(&end_time_set);
                    {
                        u64 delta_set = (u64)(end_time_set - start_time_set);
                        setup_time = (float)delta_set / COUNTS_PER_SECOND;
                        // Display load and setup time
                        printf("%8.5f\t", setup_time);
                    }

                    // Start Process
                    process_data = TRUE;
                }
            }
        }
    }


    //---------------------------------------------------------------------------
    // Cleanup Platform
    //---------------------------------------------------------------------------
    cleanup_platform();

    //---------------------------------------------------------------------------
    // Exit Application
    //---------------------------------------------------------------------------
    return 0;
}
```